Finanziato dall'Unione europea
NextGenerationEU

Ministero dell'Università e della Ricerca

Italiadomani
PIANO NAZIONALE DI RIPRESA E RESILIENZA

ICSC
Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

Recent Advances on PLUTO GPU Development and Astrophysical Applications

A. Mignone[*]

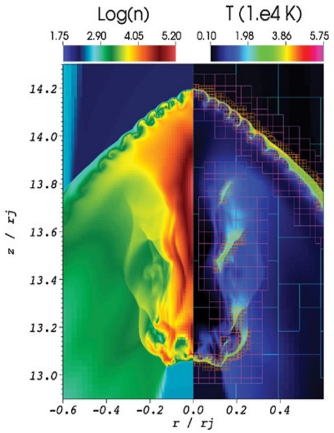
Collaborators: A. Suriano[*]*, M. Rossazza*[*] *, S. Truzzi*[*]*, V. Berta*[*]*, M. Bugli*

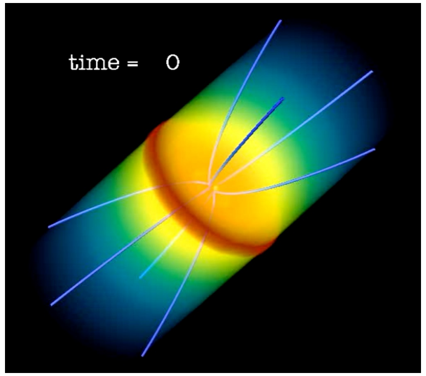
[*] Università di Torino

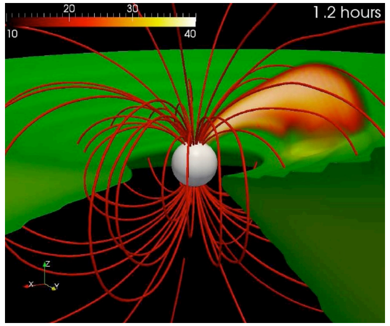
Spoke 3 General Meeting, Elba 5-9 / 05, 2024

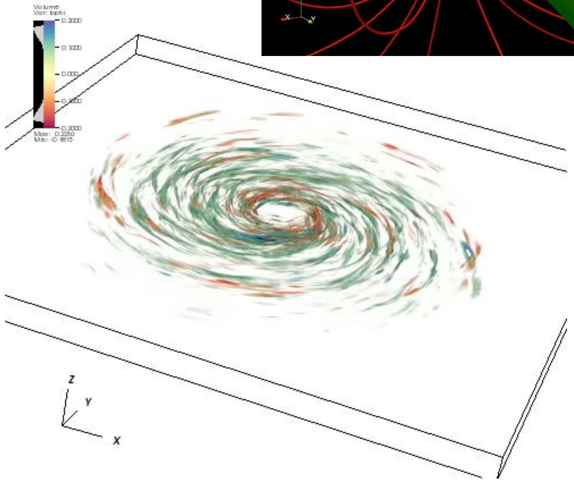
What is PLUTO ?

- PLUTO[1,2] is a finite volume (FV) Godunov-type, fluid-particle hybrid code for plasma dynamics in astrophysics;
- Target: multidimensional compressible fluid / plasma with large Mach numbers;
- Multiphysics modular support: classical fluid dynamics → special relativistic MHD;
- Non-ideal physics: viscosity, thermal conduction, resistivity, heating, etc…
- Algorithm modularity: combination of different numerical schemes;
- Publicly available at http://plutocode.ph.unito.it (v. 4.4 – CPU version)









---

[1] Mignone et al. ApJS (2007), 170, 228-242; [2] Mignone et al, ApJS (2012), 198, 7

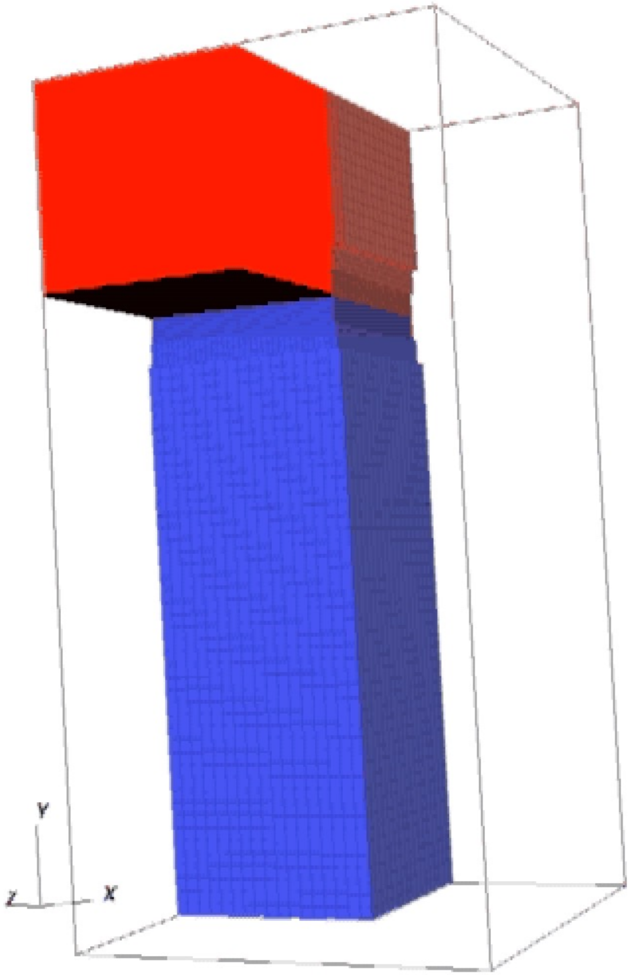
Fluid Equations, Finite Volume

- PLUTO is (primarily) an Eulerian code, solving conservation laws on a fixed / adaptive grid, e.g.:

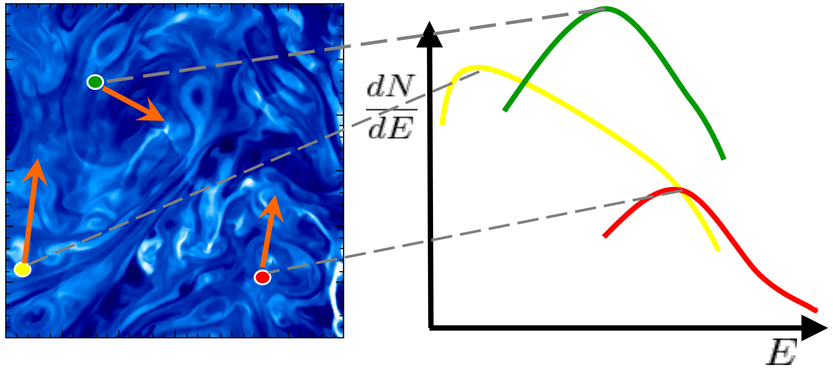
$$
\boxed{
\begin{array}{lcll}
\dfrac{\partial \rho}{\partial t} + \nabla\cdot(\rho\mathbf{u}) & = & 0 & \text{(Mass cons.)} \\
\dfrac{\partial (\rho\mathbf{u})}{\partial t} + \nabla\cdot\left[\rho\mathbf{u}\mathbf{u} - \dfrac{\mathbf{B}\mathbf{B}}{4\pi} + \left(p + \dfrac{\mathbf{B}^2}{8\pi}\right)\right] & = & 0 & \text{(Momentum cons.)} \\
\dfrac{\partial E}{\partial t} + \nabla\cdot\left[\left(E + p + \dfrac{\mathbf{B}^2}{8\pi}\right)\mathbf{u} - \dfrac{(\mathbf{u}\cdot\mathbf{B})}{4\pi}\mathbf{B}\right] & = & 0 & \text{(Energy cons.)} \\
\dfrac{\partial \mathbf{B}}{\partial t} + \nabla\cdot(\boldsymbol{u}\boldsymbol{B} - \boldsymbol{B}\boldsymbol{u}) & = & 0 & \text{(Mag. flux cons.)}
\end{array}
}
$$

- Shock-capturing relies on FV formalism, where equations are solved using the integral representation:

$$\frac{d\,\langle U\rangle}{dt} = -\oint \mathbf{F}\cdot d\mathbf{S}$$



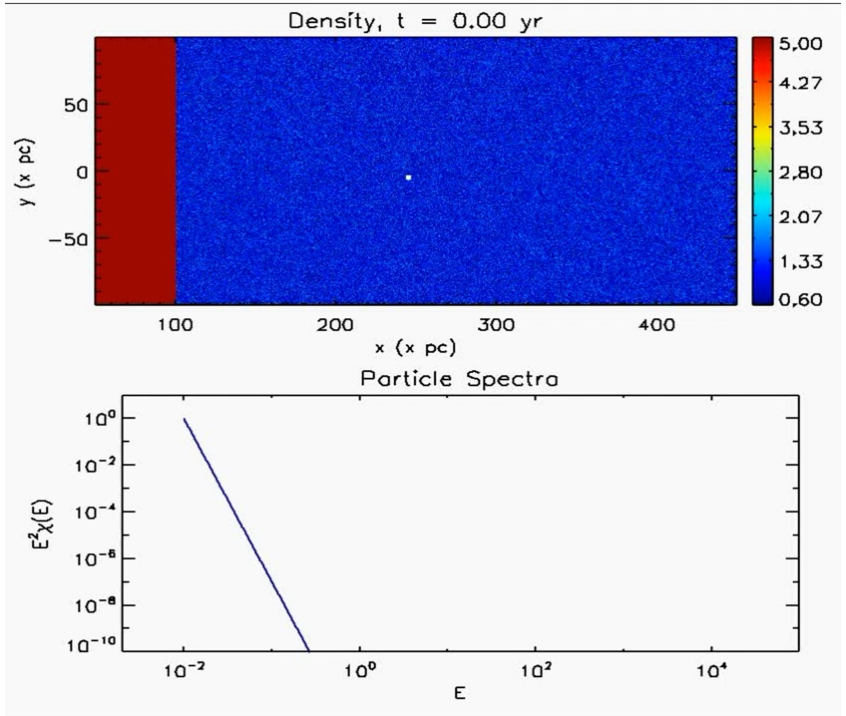
Hybrid Fluid – Particles Methods



- Target: Large-scale non-thermal emission from high-energy sources.
- Lagrangian Particles (LP)[1]: Ensemble of electrons close in physical space, characterized by a distribution function $f = dN/dE(\varepsilon, t)$ representing the actual particle number density as a function of energy ε.
- LP are transported at the fluid speed ($dx/dt = v_g$) but their spectra is evolved by solving, for each particle, a Fokker-Planck equation:

$$\nabla_\mu(u^\mu f_0 + q^\mu) + \frac{1}{p^2}\frac{\partial}{\partial p}\left[-\frac{p^3}{3} f_0 \nabla_\mu u^\mu + \langle \dot{p}\rangle_l f_0 - \Gamma_{\rm visc} p^4 \tau \frac{\partial f_0}{\partial p} - p^2 D_{pp}\frac{\partial f_0}{\partial p} - p(p^0)^2 \dot{u}_\mu q^\mu\right] = 0$$

Single LP, Fermi I Shock Acceleration



---

[1] Vaidya et al. ApJS (2018), 865, 144V

PLUTO (CPU version):

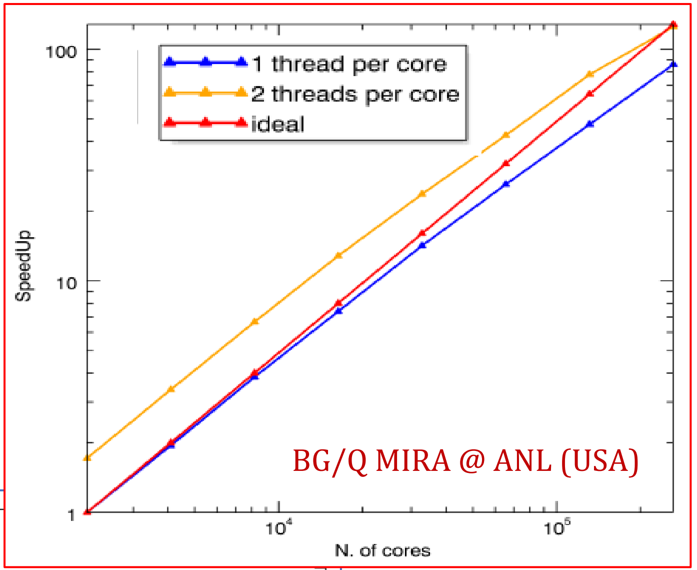
- Written in C (~110,000 lines) and C++ (6,000 lines) and python (user interface);

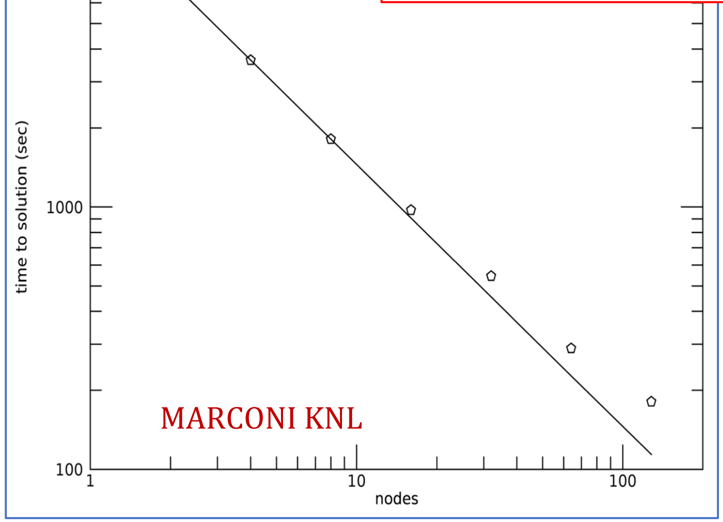
- Supports single- and multi-core parallel computations through the MPI library. Tested up to up to 262,144 cores and several different platforms.

- Computations may be performed on

- Static grid : single fixed grid (library free);

- *Adaptive grid:* multiple refined, block-structured nested grids (CHOMBO Lib)





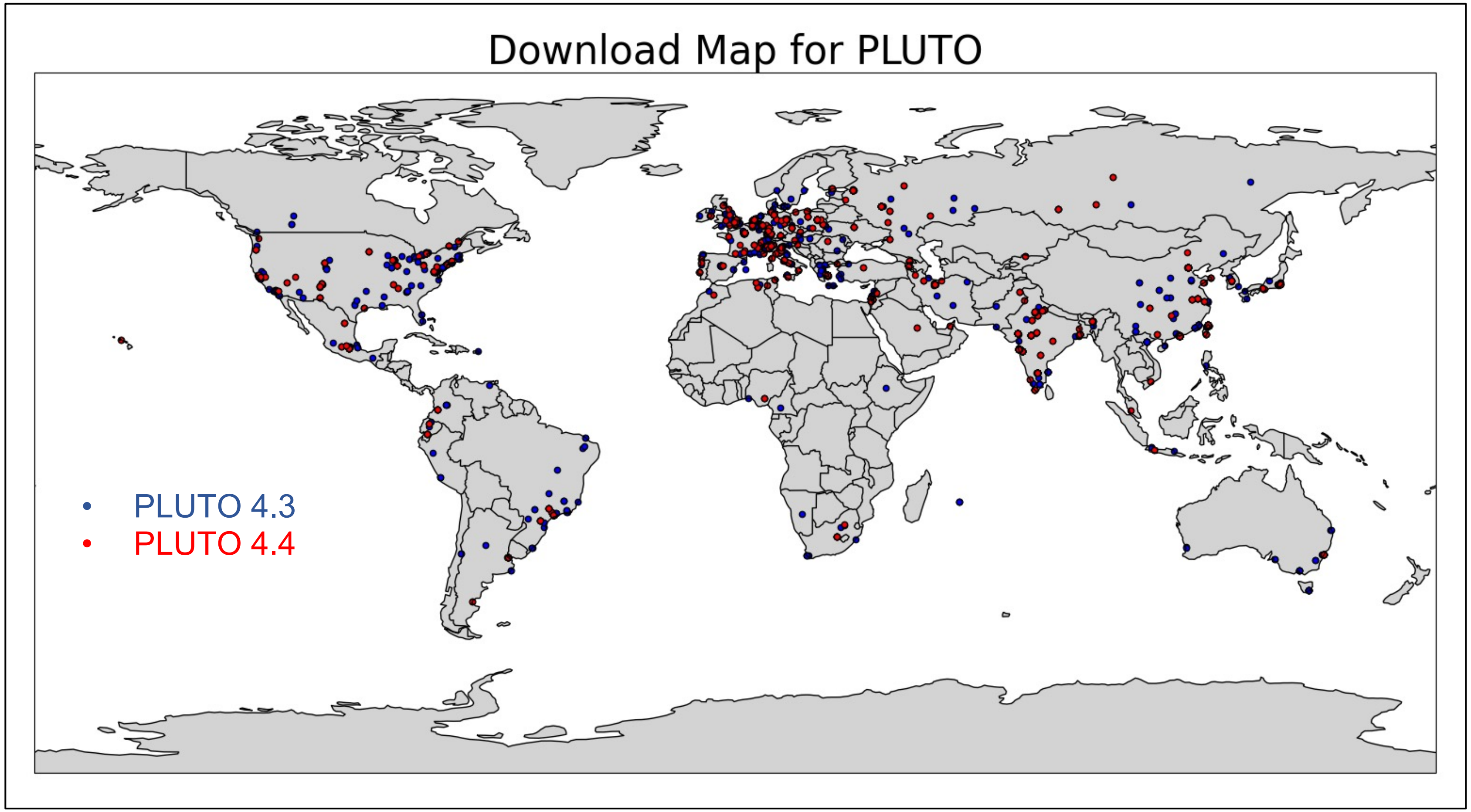
# Nodes	Time to sol (sec)	Speedup
2	*7260*	*2*
4	*3630*	*4*
8	*1815*	*8*
16	*974*	*14.9*
32	*545*	*25.6*
64	*289*	*50.2*
128	*181*	*80.22*

PLUTO Worldwide Distribution

- Heterogeneos application domain: Planet Formation / Stellar & extragalactis Jets / Radiative shocks / accretion disks / Jet launching / magnetospheric accretion / Jet star interaction / Plasma instabilities (MRI, KHI, CDI, RTI, etc…)

- PLUTO 4.3, (2018-2021) ~ 1360 downloads

- PLUTO 4.4, (2020-2021) ~ 460 downloads



Objectives: GPU Porting + Revision Process + Public release

Aims:

1. Exhaustive **porting** of the code to GPU;

2. Complete Code **revision** (PLUTO is 18 years old !);

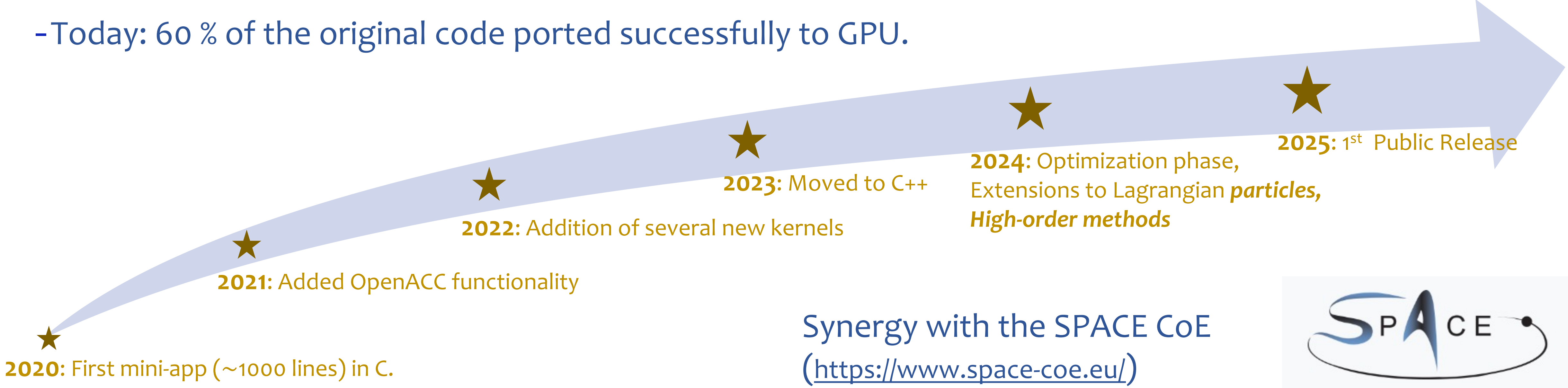
3. Public **release** (→ "**gPLUTO**").

- Roadmap started in 2020, → **full code rewrite** + NVIDIA support [except for a few kernels, e.g. initialization, I/O, user interface, etc…];

- **C++** & **OpenACC** (a high-level directive based programming model developed by NVIDIA) chosen as our programming paradigm.

Activities Timeline

- Code rewritten from scratch (!) in 2020: with simple HD module (miniapp, ~1000 lines);
- Incrementally added modules & kernels;
- Switched to C++ to exploit more versatile construct (e.g., templates, classes, vectors);
- Today: 60 % of the original code ported successfully to GPU.



2020: First mini-app (~1000 lines) in C.

2021: Added OpenACC functionality

2022: Addition of several new kernels

2023: Moved to C++

2024: Optimization phase, Extensions to Lagrangian ***particles, High-order methods***

2025: 1st Public Release

Synergy with the SPACE CoE
(https://www.space-coe.eu/)

OpenACC: Basic Facts

- Why OpenACC ? → i) high-level, ii) requires few changes to the code, iii) directive-based;

- Two main directives or pragmas: i) compute pragmas & ii) data pragmas.

- The `#pragma acc parallel loop` directive indicates that a loop can be parallelized and executed in parallel on the GPU:

```
#pragma acc parallel loop vector
for (i = 0; i < N; i++){
  // Loop body
  // ... Things to do here ...
}
```

- The `#pragma enter data copying` directive explicitly transfers data from the CPU memory to the GPU memory.

```
#pragma acc enter data copyin (A,B)
// ... Code where A ad B are used in GPU computations
#pragma acc exit data delete (A,B)
```

OpenACC: Keypoints

Data Locality: reduce data movement between CPU and GPU memory as much as possible.

Data transfer *major bottleneck* → solution straightforward: all computational part of the program should reside in GPU memory !

Private Variables: GPU threads should perform identical operations but on different memory addresses.

Without precautions, simultaneous operations are performed at the same memory address leading to incorrect results.
→ Private variables have local scope and are allocated individually for each thread.

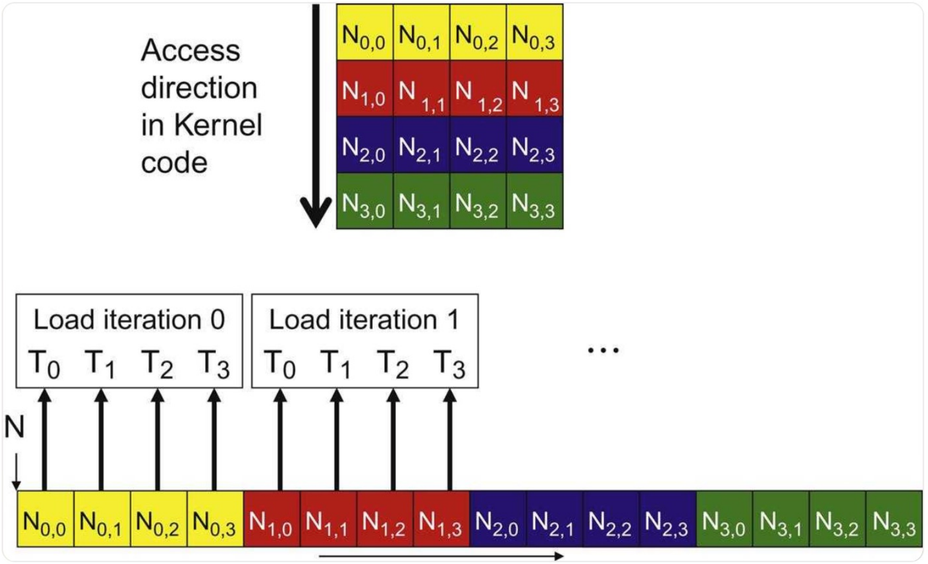
```
int V[8];
int A[NX][NY][NZ];

#pragma acc parallel loop collapse(3) private(V[:8])
for (i = 0; i < NX; i++){
for (j = 0; j < NY; j++){
for (k = 0; k < NZ; k++){

  A[i][j][k] *= 2.0;

  V[0] = ...;
  V[1] = ...;
  ...
}}}
```

OpenACC: Keypoints

<u>**Coalesced Memory Access**</u>: consecutive threads access consecutive memory addresses. Memory coalescing is a technique which allows optimal usage of the global memory bandwidth.

→ the GPU can perform memory transactions more efficiently, reducing the overall memory access time and improving performance.

→ Requires *<u>different</u>* array ordering so that the inner loop we're accelerating should be also the fastest index of the multidimensional array as in this example.



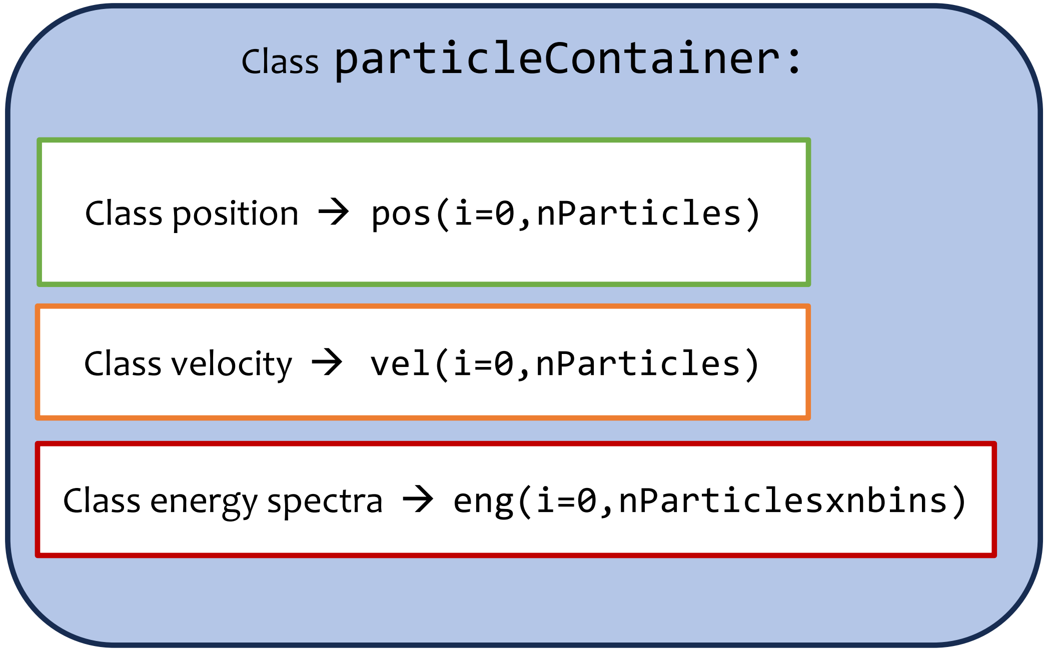
```
#pragma acc parallel loop vector
for (i = ibeg; i <= iend; i++){
  #pragma acc loop seq
  for (nv = 0; nv < NVAR; nv++) {

    // MUST REVERSE INDICES HERE:

    v[i][nv] *= 2;   →   v[nv][i] =*= 2;
  }
}
```

<u>**Using C++ templates:**</u> `v[nv][i]` → `v(i,nv)`

OpenACC: Particles

- Particle are constantly injected and deleted.
- Previous versions (PLUTO 4.4) based on linked list.
- Problem: linked list not easy parallelizable on GPU !

→ Need to go back to arrays → Classes (C++)



Class `particleContainer:`

Class position → `pos(i=0,nParticles)`

Class velocity → `vel(i=0,nParticles)`

Class energy spectra → `eng(i=0,nParticlesxnbins)`

- Parallelizeble structure, e.g.:

```
std::vector<double*> pos;
for(int i = 0; i < nChunks; i++){
  pos.push_back( new double[chunkSize] );
}
```

- Reshaping memory is expensive: memory allocation in chunks:

```
#pragma acc parallel loop present(pc)
for (i = 0; i < pc.Size(); i++){
  partContainer.pos(i) = 4.56;
}
```

Results: MHD 3D

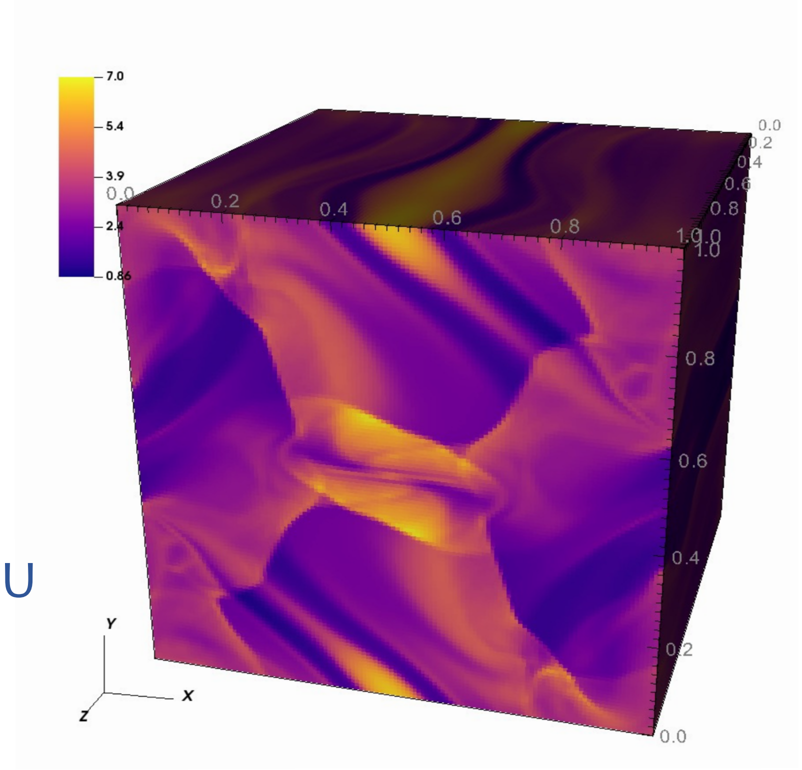
- Weak scaling on the 3D version of the Orszag-Tang vortex;

$$\mathbf{v} = -\zeta(z)\sin(2\pi y)\hat{\mathbf{e}}_x + \zeta(z)\sin(2\pi x)\hat{\mathbf{e}}_y + 0.2\sin(2\pi z)\hat{\mathbf{e}}_z$$

$$\mathbf{B} = -B_0\left[\sin(2\pi y)\hat{\mathbf{e}}_x + \sin(4\pi x)\hat{\mathbf{e}}_y\right]$$

- where $\zeta(z) = 1+\sin(2\pi z)/5, B_0 = 1/\sqrt{4\pi}$

- Scaling conducted on Leonardo equipped nodes with Intel Ice Lake CPU and 4 NVIDIA A100 ("Da Vinci" variant) up to 256 nodes (= 1024 GPU).

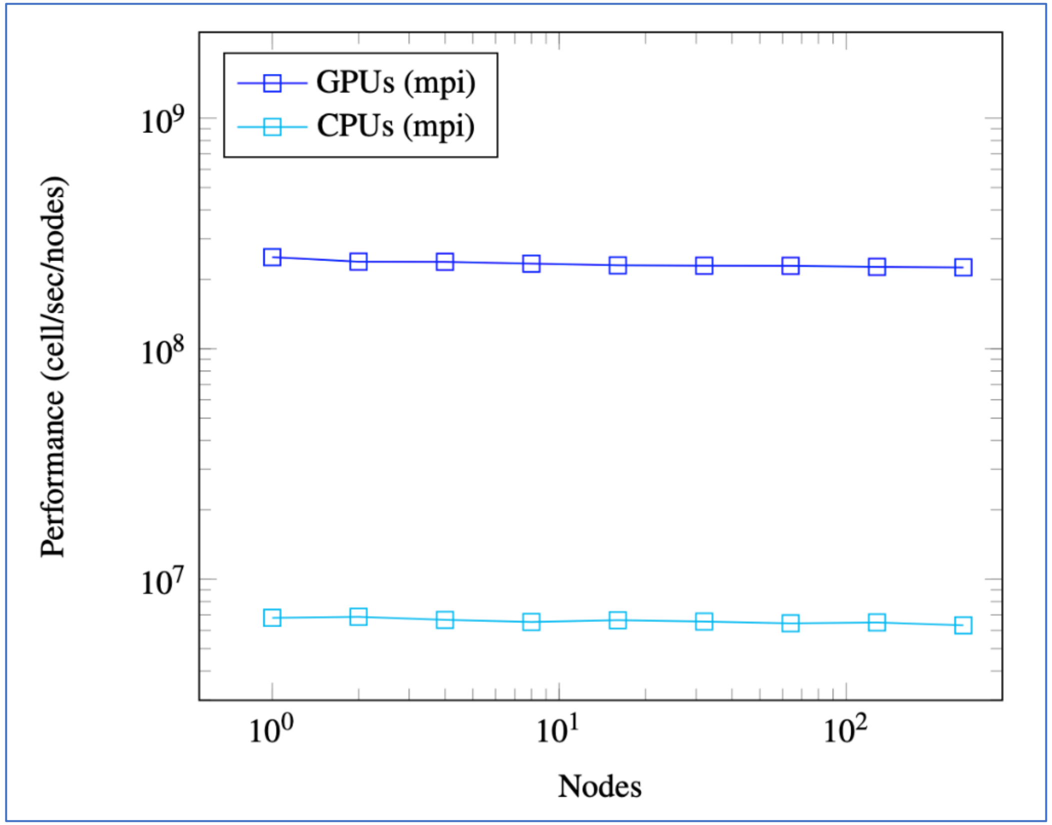


- **Weak scaling** (640^3 grid cells per node) using 3 different configurations
  1. CPU + MPI / 2. GPU + MPI / 3. GPU + NCCL

Results: 1) CPU-GPU Speedup

Nodes	$T_{GPUs_{nccl}}$ (sec)	$T_{CPUs_{mpi}}$ (sec)	Acceleration (T_{CPUs}/T_{GPUs})
1	466.8	15696.8	33.62
2	483.5	15492.1	32.04
4	496.2	15928.0	32.09
8	518.8	16212.3	31.25
16	549.8	15905.6	28.93
32	570.5	16096.1	28.21
64	558.7	16356.0	29.28
128	583.7	16199.6	27.75
256	586.5	16659.2	28.40

Execution time of the weak scaling tests for 400 steps. A speed-up factor in the ≈ 28.4 – 33.6 range is measured.



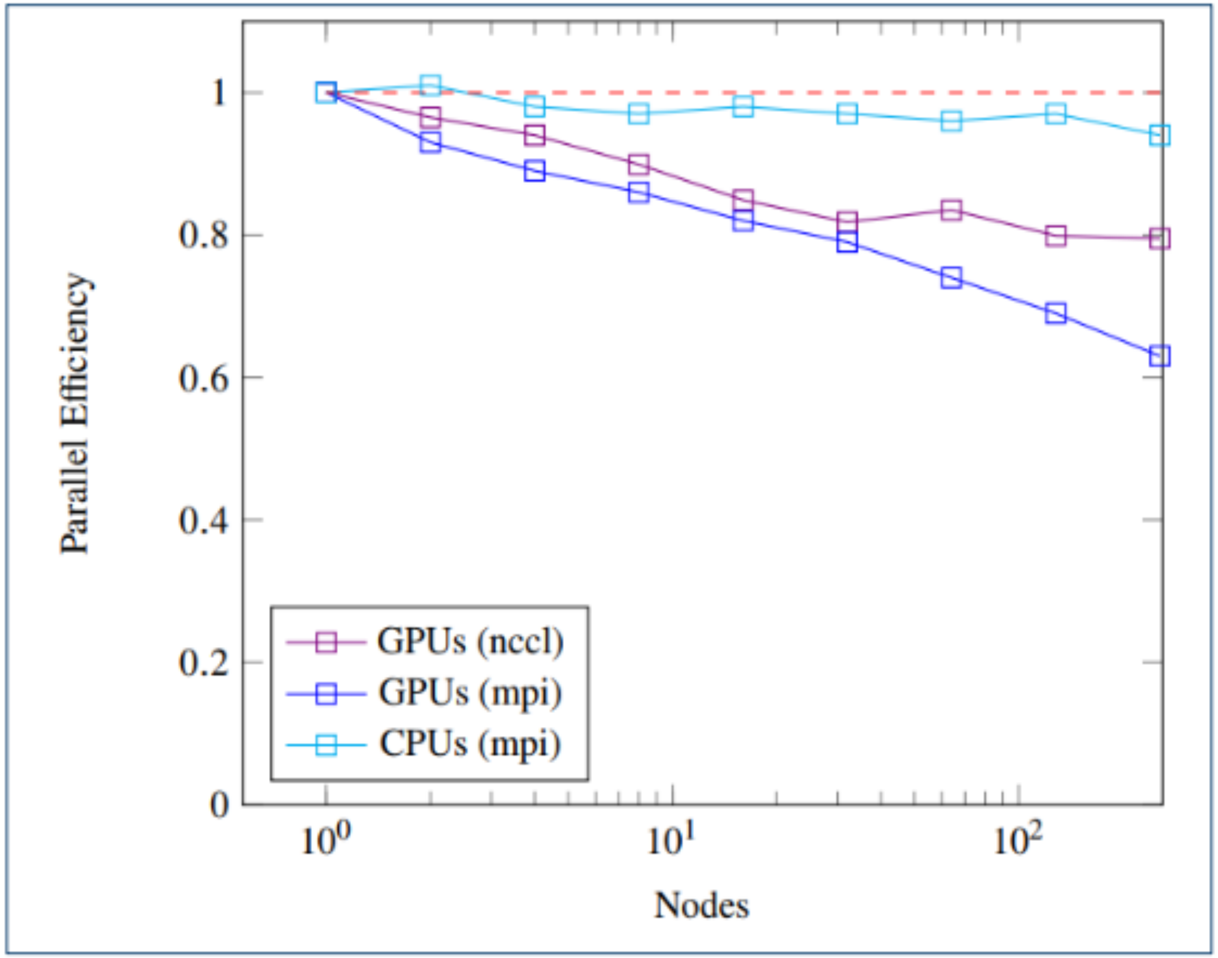
In the figure, the values represent the number of steps and grid points handled by each node.

Results: 2) Weak Scaling (Synchronous version)

Version #1: synchrounous Send/Recv calls:

```
// Fill buffer
send_bufL[] ← data()
send_bufR[] ← data()

// Send / Receive data
MPI_Sendrecv (send_bufL, count, MPI_DOUBLE, procL,
              recv_bufR, count, MPI_DOUBLE, procR, ... )
```

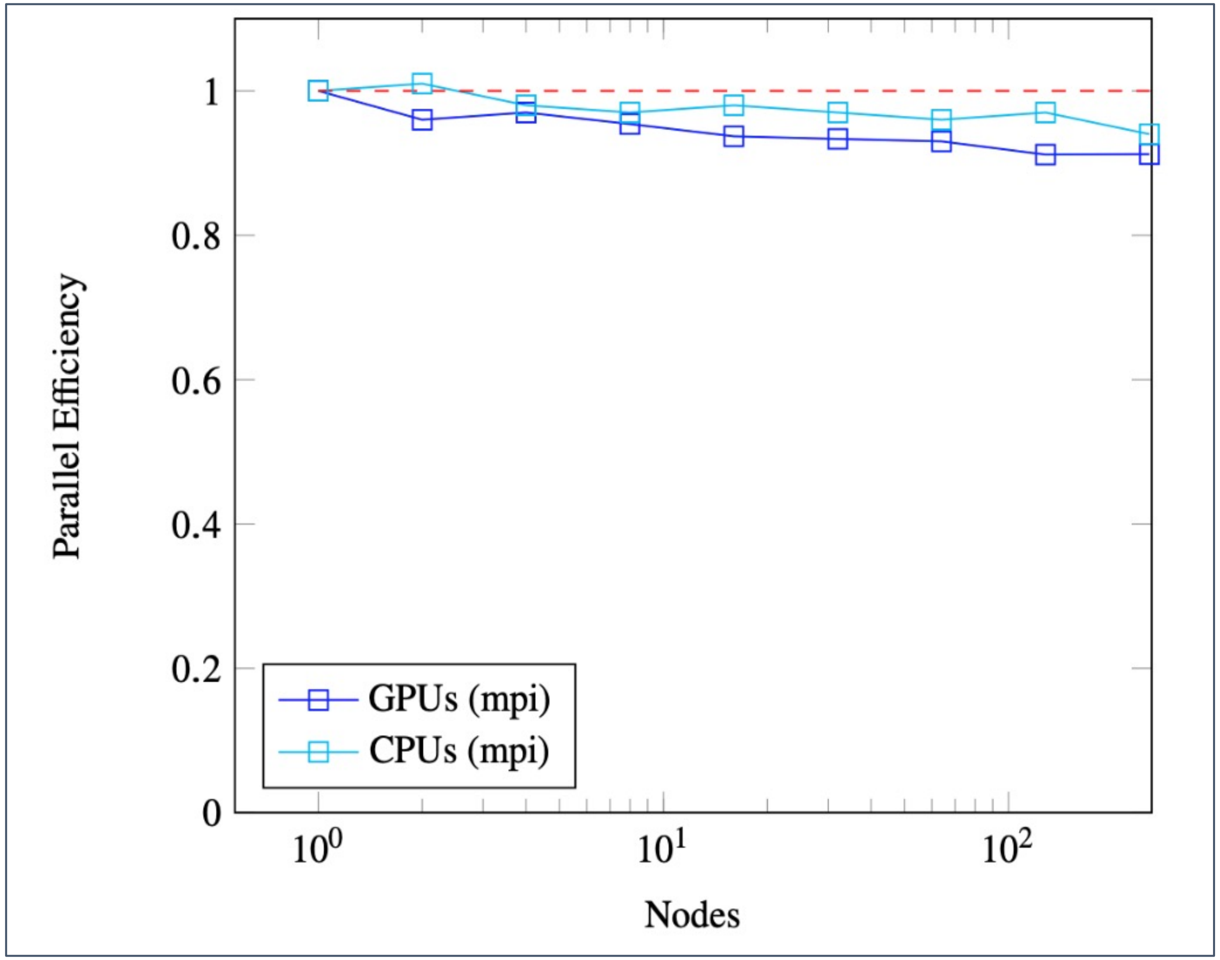
→ Not optimal for GPU computations ←



Results: 2) Weak Scaling (Asynchronous version)

Version #2: asynchrounous Send/Recv calls:

```
// Initiate ascynchrounous receive
MPI_Irecv (recv_buf, ... , recv_proc, ... , MPI_recv_req )

// Fill buffer
snd_bufL[] ← data()

// Send data
MPI_Isend (send_buf, ... , send_proc, ... , MPI_send_req)

// Wait for MPI receive request to complete
MPI_Waitall ( ... , MPI_recv_req,  ...)

// Unpack buffers
data() ← recv_buf[]

// Wait for MPI send request to complete
MPI_Waitall ( ... , MPI_send_req,  ...)
```



Next Steps and Expected Results

- Extension of asynchrounous inter-GPU communication to NCCL;
- Improving particle scaling on large number of CPUs and GPUs;
- Addition of non-Cartesian geometries;
- Addition of non-ideal terms (viscosity, thermal conduction, resistivity);
- Addition of cosmic-rays particles and dust particles;
- Adaptive Mesh Refinement.

THANK YOU