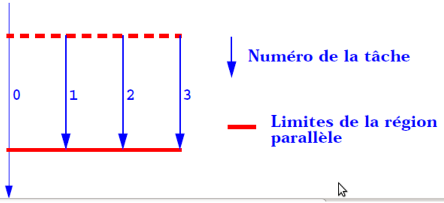
OpenMP parallelization

```
!$ Initialisation des données
  ...
!$OMP PARALLEL [IF (M > 512)] [clause1 [, clause2] ... ]
  ... région parallèle
!$OMP END PARALLEL
```



OpenMP parallelization

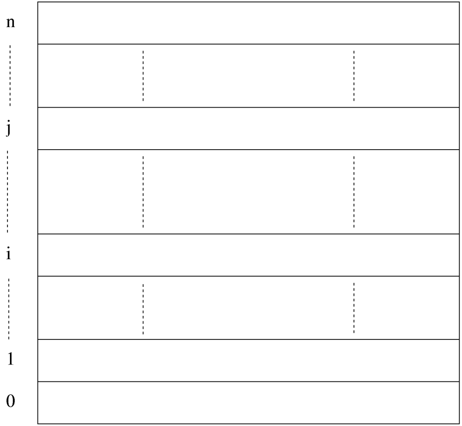


FIGURE: x-step : each thread computes a slice

OpenMP Performances, Sod shock tube

- Tests on Titane (CCRT) : processor Intel Xeon 5570 quadri-coeurs

Number of threads	1	2	4
CPU time (s)	530	310	230
Speed up		1.70	2.30
Efficiency		0.86	0.58
Cells/thread	$25.6K$	$12.8K$	$6.4K$

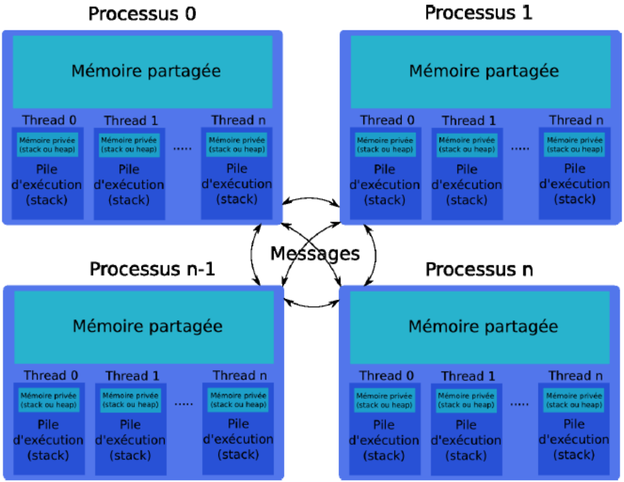
TABLE: Mesh 160 $\times$ 160

Summary of the OpenMP parallelization

- A big restructuring of the sequential code was necessary to obtain an efficient OpenMP version ;
- This restructuring allows to use other pragma directives like HMPP ;
- One this restructuring done, incremental OpenMP parallelization is possible and quite easy. If we compare the source codes of the sequential implementation to the OpenMP approach there is not much difference ;
- The results do not depend on the threads number ;
- The pragma directives allow to keep only one verion of the code ;
- This OpenMP algorithm remains efficient under 10 000 cells per core ;
- A OpenMP execution is restricted to only one shared memory node ;
- The openMP part is decoupled of the MPI communications part, which makes easier the development of a hybrid MPI+OpenMP version ;
- More intrusive developements in the numerical scheme compared to the MPI version, but less intrusive compared to other languages like Pthreads or TBB.

Hybrid parallelization MPI+OpenMP

Hybrid architecture



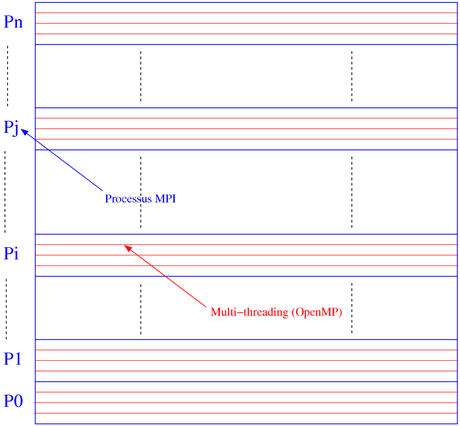
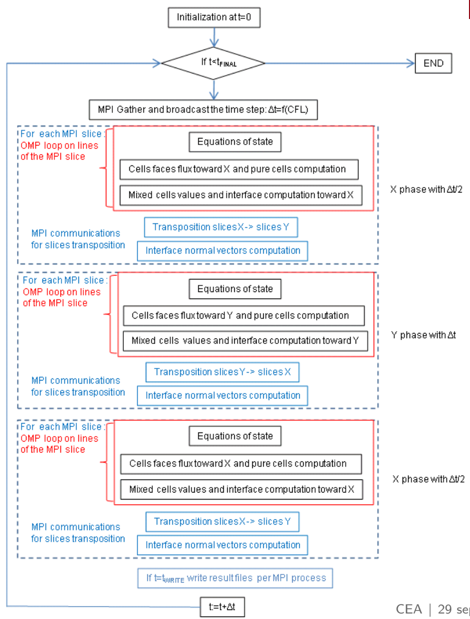


FIGURE: x-step : Multi-threading (with OpenMP) in each slice Pi

- We associate the MPI slice decomposition with the OpenMP parallelization (distributed memory between slices and shared memory into the slice).
- Each sub-domain (slice) Pi is computed by a processor or a node denoted by $proc_i$ (for instance processeur Intel Xeon 5570 quadri-coeurs for Titane) and MPI communications (transposition) are done between these $proc_i$ as in the full-MPI parallel algorithm (distributed memory).
- In each slice (shared memory, for a given $proc_i$), the previous OpenMP parallelization is used.

Algorithm description



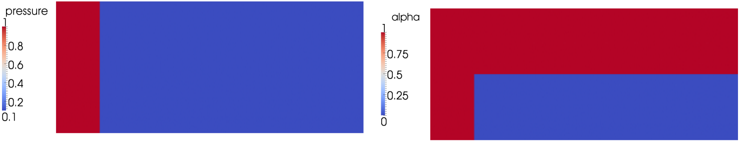


FIGURE: Pressure (left) and volume fraction (right) at initial time, triple point shock tube.

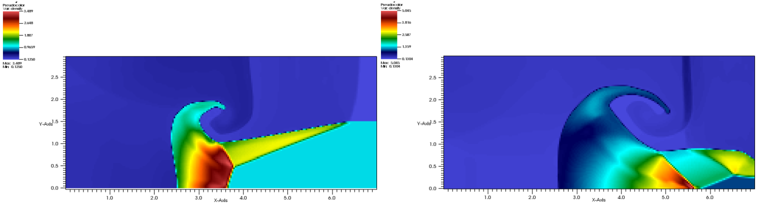


FIGURE: density at time 3.3 (left) and density at final time 5 (right), triple point shock tube with mesh 210 $\times$ 90.

Performances, triple point shock tube

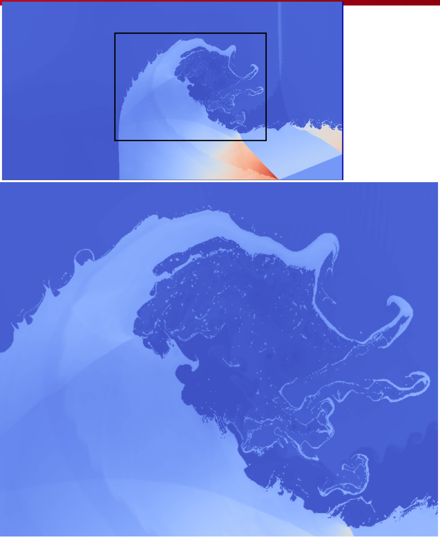


FIGURE: Density full geometry (up) and zoom on small structures (down), mesh 6144x2048 triple point shock tube.

Performances, triple point shock tube

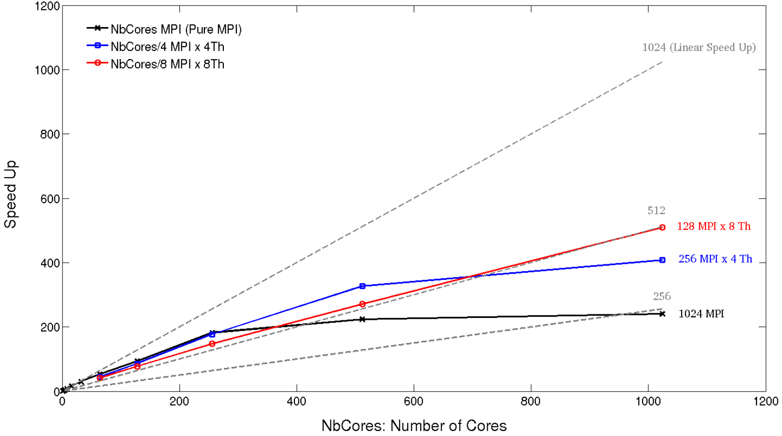


FIGURE: Strong scalability using Pure MPI and MPI+OpenMP (8M cells, 5000 iterations).

Performances, triple point shock tube

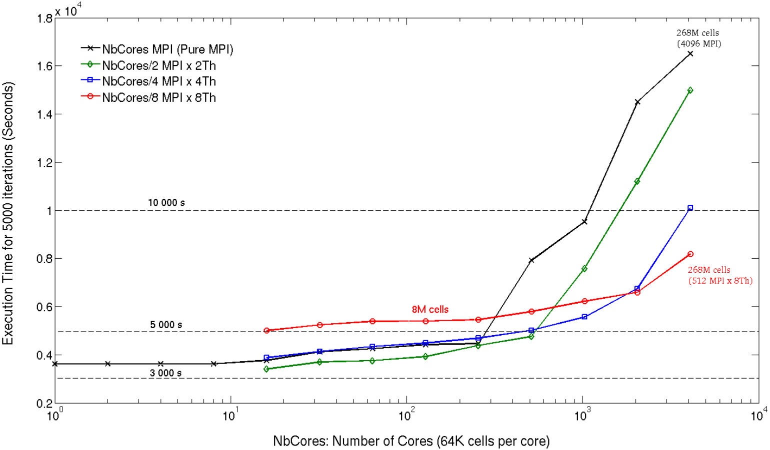


FIGURE: Weak scalability using different parallelization strategies (different number of OpenMP threads per MPI process). 64K cells per core, 5 000 iterations.

Scalasca analysis

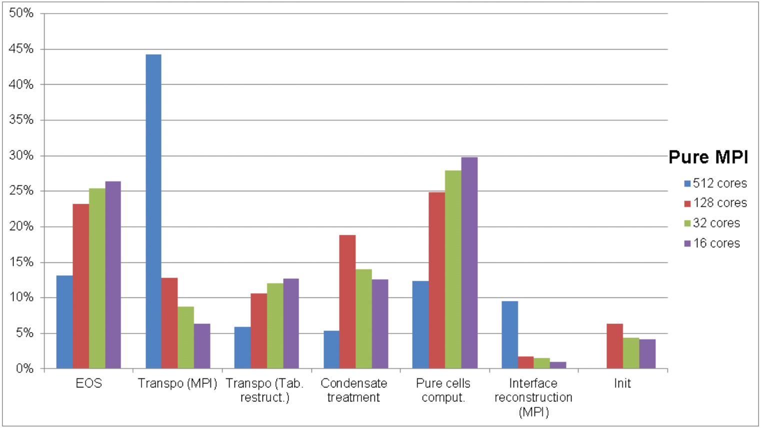


FIGURE: Percentage time spent in important FluxIC subroutines for different number of cores (8M cells, 5000 iterations). Pure MPI version.

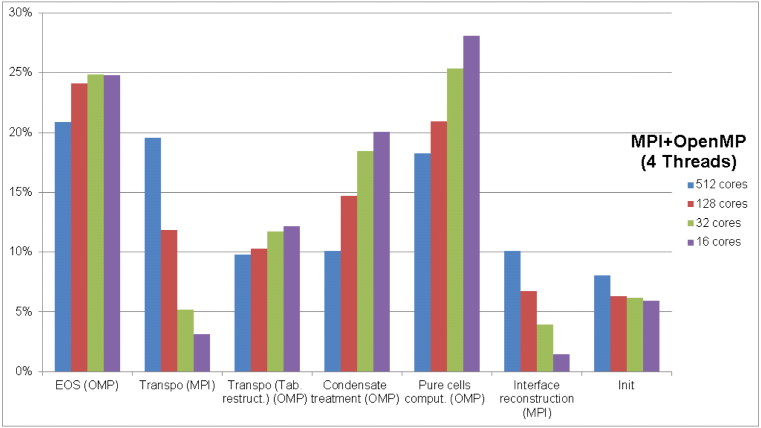


FIGURE: Percentage time spent in important FluxIC subroutines for different number of cores (8M cells, 5000 iterations). MPI+OpenMP version (with 4 threads).