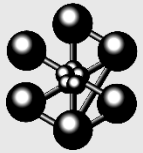
DE LA RECHERCHE À L'INDUSTRIE

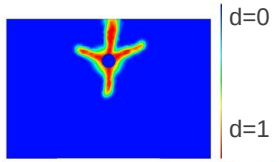
MANTA – the CEA's future platform for simulations in mechanics

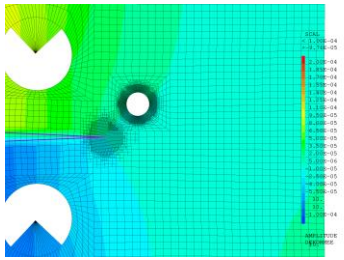
Codes et calculs implicites / explicites

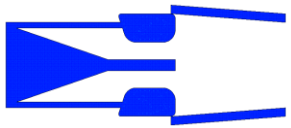
Méthodes et codes implicites/explicites

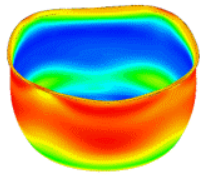
	Méthodes implicites	Méthodes explicites
Principaux problèmes traités	Quasi-statique, dynamique lente, vibration	Dynamique rapide
Stabilité	Inconditionnellement stables 	Pas de temps critique (petit!) de stabilité 
Robustesse pour problèmes fortement non-linéaires	Faible 	Importante 
Code de calcul de mécanique développé au CEA	Cast3M 	

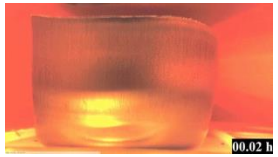
Cast3M: domaines d'application

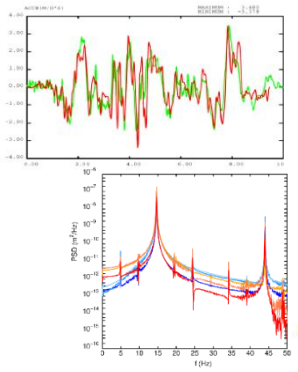


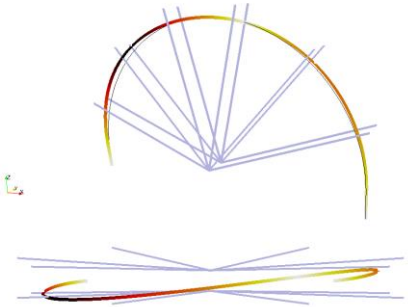


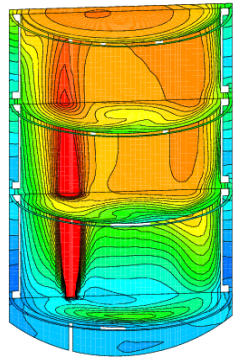


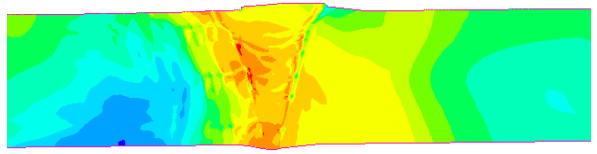


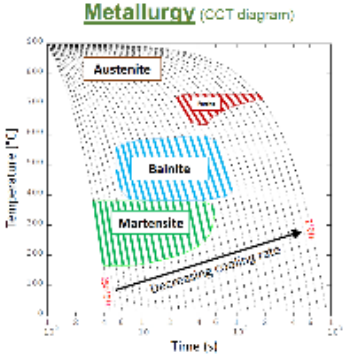












Cast3M: bilan

Forces

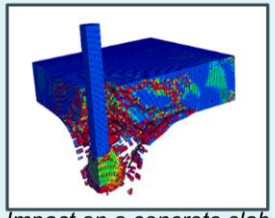
- Souple et évolutif (lié à la structuration type « boite à outils »)
- Solution tout-en-un : maillage + résolution + post-traitements
- Large spectre d'applications (vastes acquis capitalisés)
  toujours en expansion (simplicité de développement)
- Robustesse des algorithmes

Faiblesses

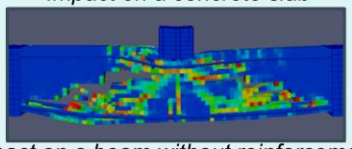
- Performances limitées (parallélisation en mémoire partagée uniquement)
  → qqs Mdofs max
- Attractivité limité du code informatique (langage, interfaçage, …)
- Information parfois difficile d'accès : notices, exemples et documentation épars

Europlexus : domaines d'application

Structure



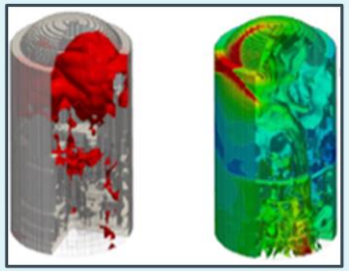
Impact on a concrete slab



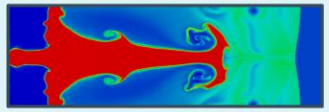
Impact on a beam without reinforcement

- **Finite Elements**
- **Erosion**
- **Damage**
- **Plasticity**
- **Sliding contact**

Fluid



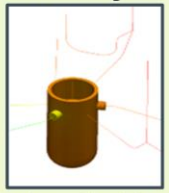
H_2 explosion in a Pressurized Water Reactor



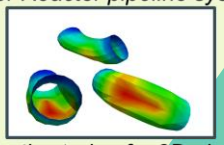
Richtmyer-Meshkov instability

- **Finite Volumes**
- **Combustion model**
- **Adaptative Mesh Refinement**

Pipeline systems



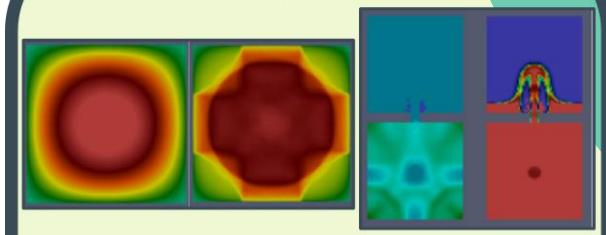
Loss of coolant accident in a Pressurized Water Reactor pipeline system



Plastic strain of a 3D pipe

- **2D/3D finite volumes**
- **1D variable cross-section elements**
- **1D/3D coupling elements**
- **Flexible pipes**

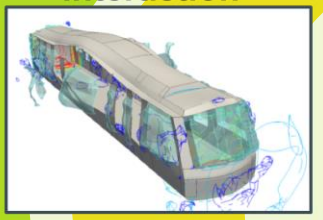
Multiphysics



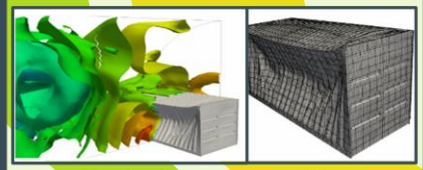
Neutronic power, pressure and liquid volume fraction in a Molten Salt Reactor (criticality accident)

- **Interoperability**
- **External and partitioned coupling**
- **MED format**
- **Coupling interface framework**

Fluid-structure interaction



Explosion in a subway train



Explosion near a container

- **Arbitrary Lagrangian Eulerian**
- **Mediating Body Method**
- **Weak and strong coupling**

epx europlexus software

Europlexus: bilan

Forces

- Développement collaboratif
  - Propriété CEA + Commission européenne
  - Consortium de développement avec EDF, ONERA et Safran Tech
- Solveur explicite efficace pour le CEA et ses partenaires
  - Modèles adaptés notamment pour l'IFS : solveurs et méthodes dédiées
  - Recherche de performance (historiquement, résolution de gros problèmes en séquentiel)

→ Répond au besoins actuels (~calcul sur 10 M éléments sur 100 proc possible), mais avec certaines limites …

Faiblesses

- Besoins plus importants notamment pour explosion, IFS, …
- Développements parallèles (mémoire distribuée) très complexes

Bilan :

- Codes qui répondent encore au besoin actuel des ingénieurs et des industriels
  - Codes pour la **recherche à finalité industrielle**
  - Evolution continue des **modèles** et des **fonctionnalités** des codes & **capitalisation**
  - Equipes expertes mais dimensionnées au plus juste (risque de perte de compétence)

- Codes qui montrent leurs limites face aux demandes des chercheurs et aux exigences du HPC
  - Confirmé par le REX des proto-applications à vocation HPC : Passmo (séisme), Helix (thermomécanique statique) et Mefisto (explicite IFS)

- Codes « **anciens** » : être conscient des **forces** et des **faiblesses**
  - Force : héritage des modèles et du savoir-faire technique, codes éprouvés → maintenance limitée
  - Faiblesse : difficile transmission des compétences (documentation), dettes techniques, stratégies numériques à rénover

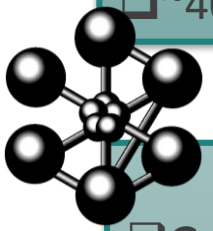
MANTA

Main objectives

2030: industrial operation

epx europlexus software

- **Explicit dynamics** for **structures** and **compressible fluids**
- **Fluid / structure interactions**
- **Industrial applications**
- **Finite-elements**, **finite-volumes**, sph, discrete element method
- ~40 years of development



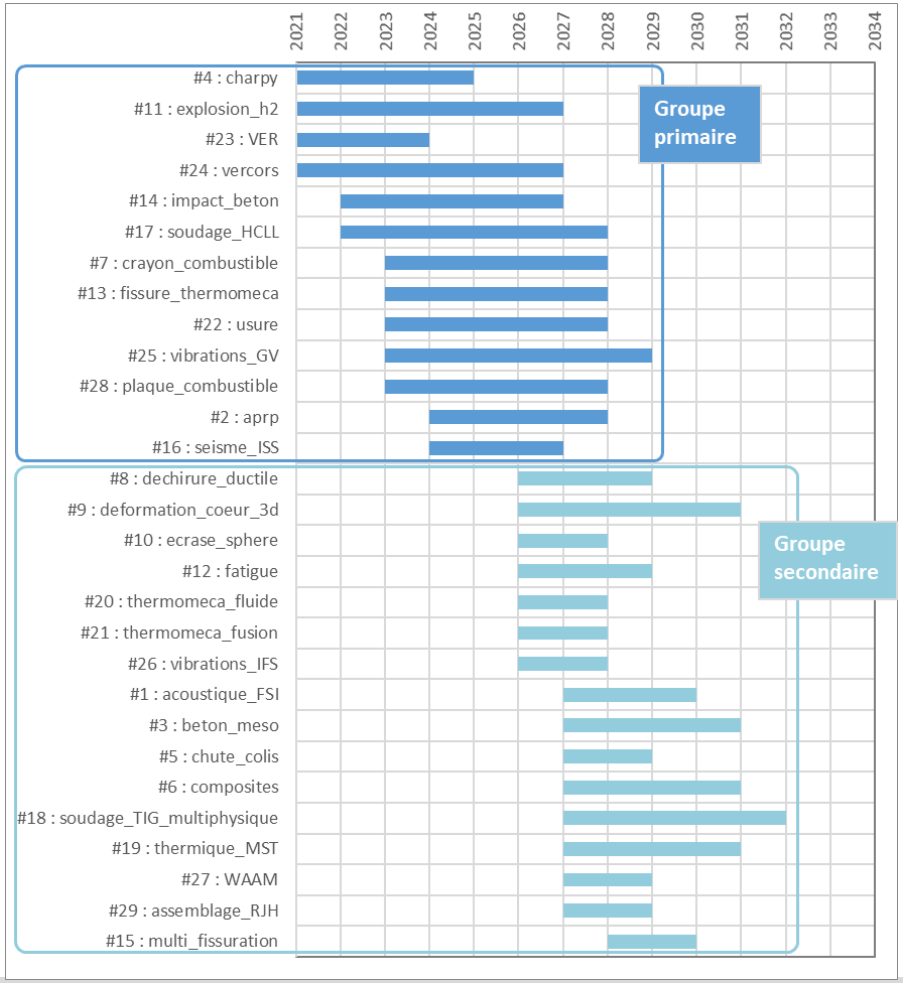
- **Generic tool** for "**implicit problems**"
- **Mainly geared for (non-linear) mechanics**
- **… but also applied to** incompressible fluids, electromagnetism, metallurgy, …
- **Industrial applications**
- **Finite-elements**
- ~40 years of development

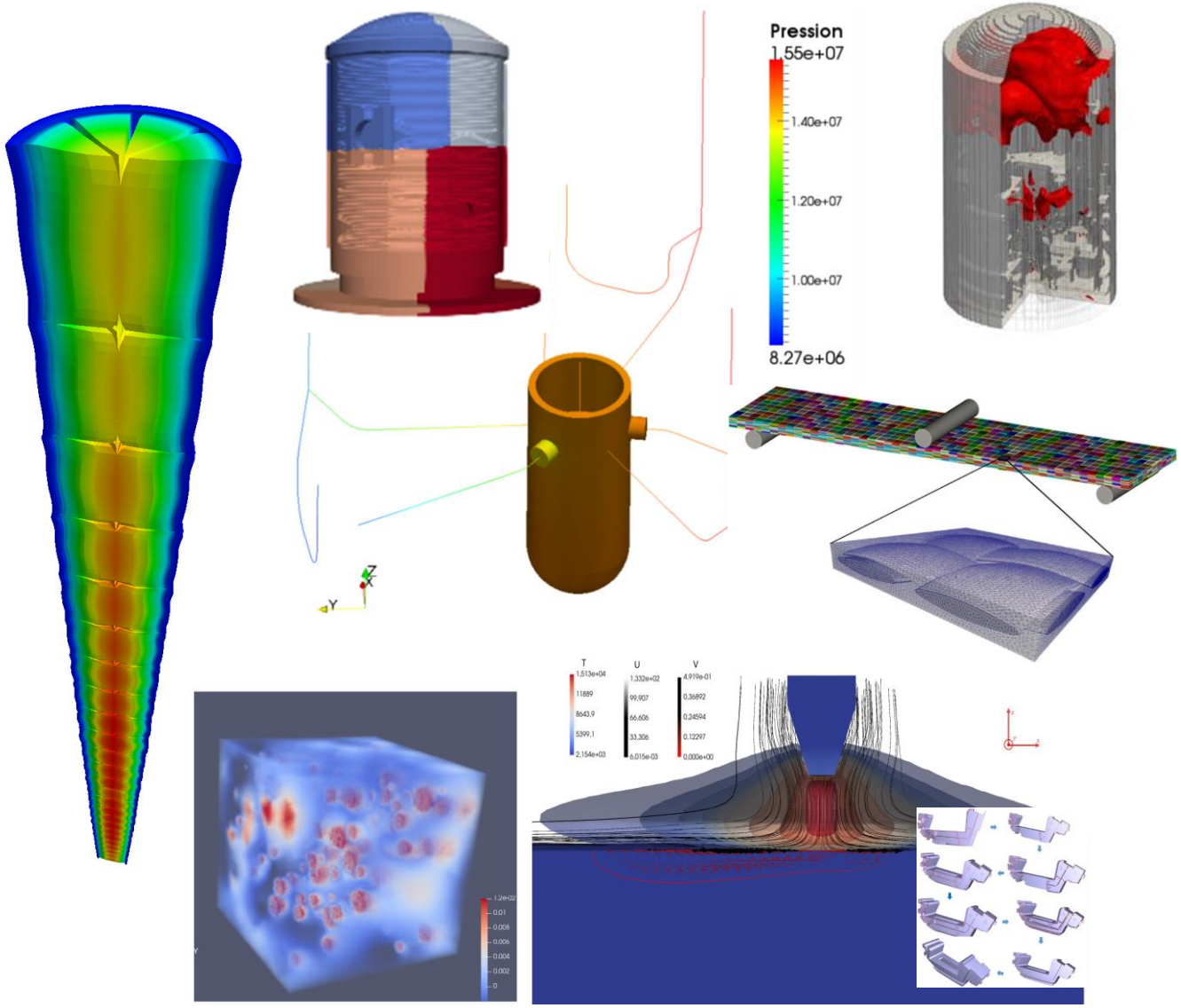
MANTA

- Next gen., **HPC** oriented
- Structure / compressible fluids / … , interactions
- Industrial applications
- Every mesh-based method (FE, FV, HDG, …)
- C++
- "automatic parallelism"
- Easy to maintain and evolve on the long term
- Open-source

From 2023, the development is driven by target industrial applications

- Evolving set and scheduling of target applications
- 29 target applications up to now

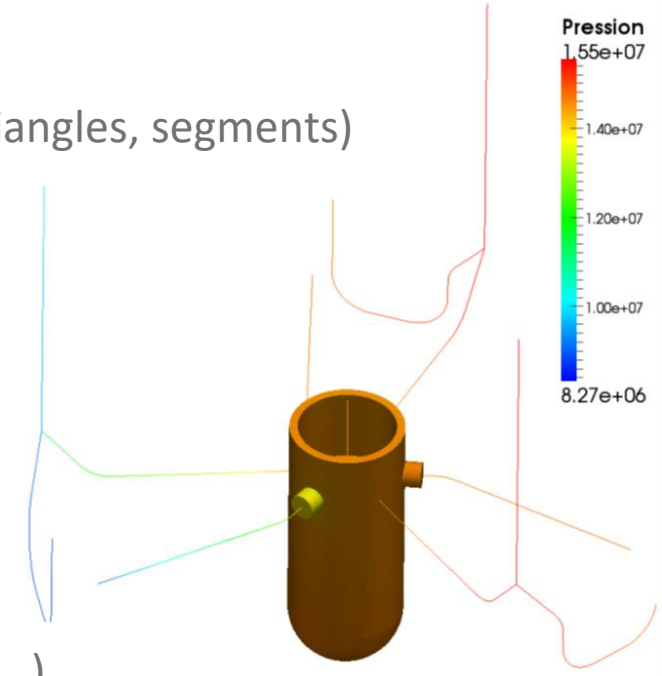




Software engineering objectives

Target industrial applications

- Multi PDE
- Lagrangian, Eulerian, ALE approaches
- Multi “areas” (more general than “multi-material”: may overlap, not cover the whole mesh, …)
- Multi topological dimension (Volume, shell, beam elements in a single calculation)
- Various geometrical supports (tetrahedral, hexahedra, prims, pyramids, quadrangles, triangles, segments)
- Very high “flexibility”, which may affect performances



HPC

- Native distributed parallelism
- Total distribution of the data, workload
  - No specificity of the process 0
  - No array of size O(global numerical model size)
- Performance portability: ability to adapt to various hardware architectures (GPU, ARM, …)

“Automatic parallelism”

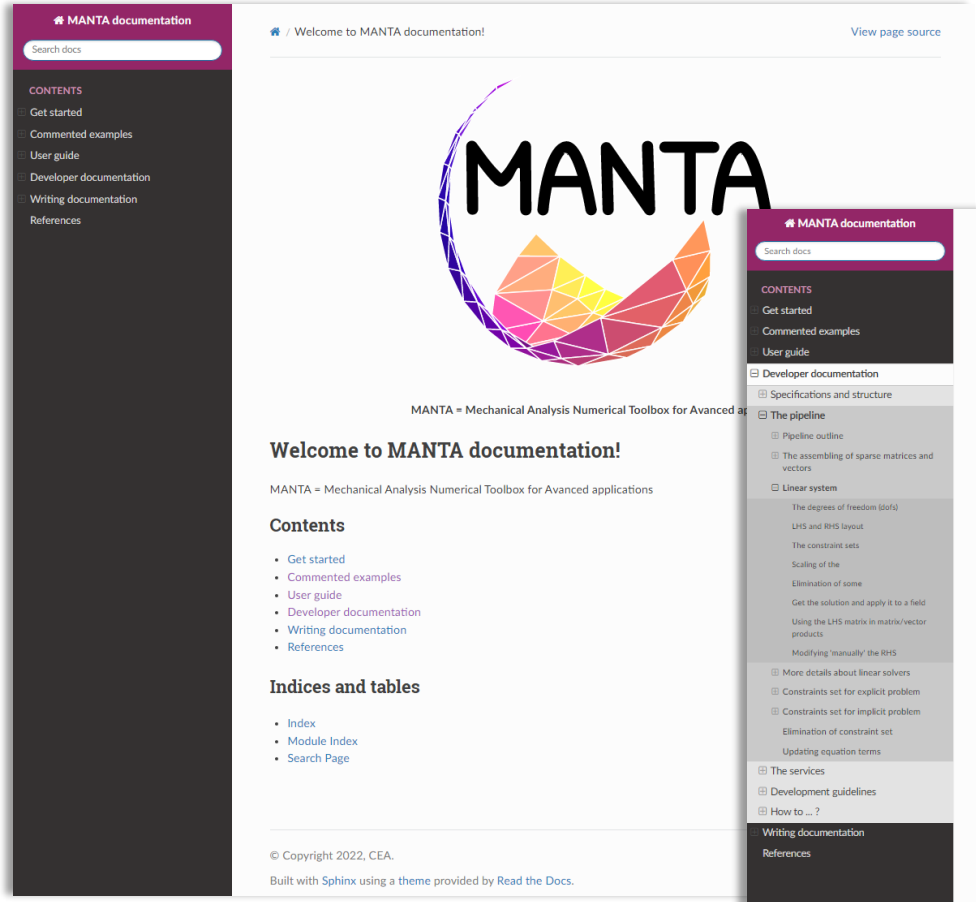
- Code new functionality “as in a sequential code”, and works in //

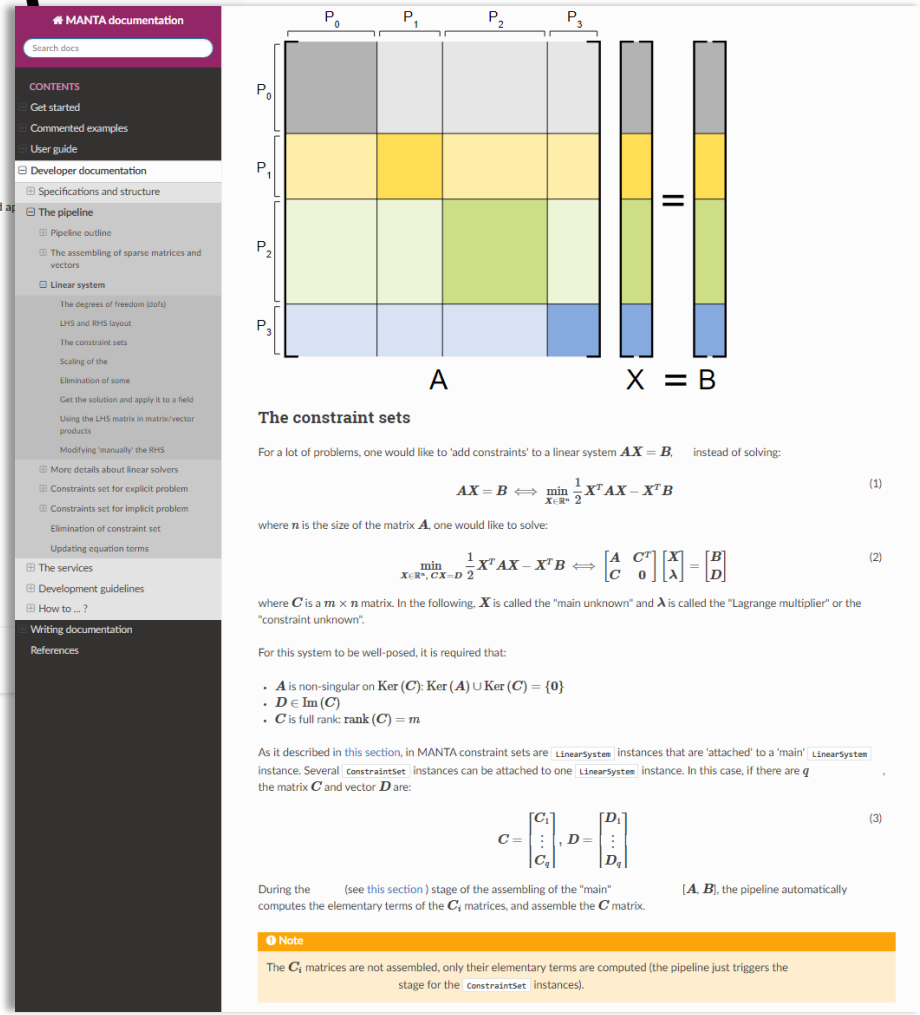
Software engineering objectives

- **Keep a low technical debt -> Easy to maintain and evolve**
  - Factorization
    - Code
    - Numerical methods -> generic “pipeline”
  - modularity
    - Clean and “localized” interfaces to 3rd parties
    - “Atomic” usage of 3rd parties functionalities
  - Rigorous collaborative workflow
    - Test battery
    - Trackers, merge requests, code review, ….
- **APIs**
  - C++ expert / user
  - Python user
  - Derive some monolithic (not a lib) executables for specific applications with dedicated input files
- **Open-source**
  - GPL (-> LGPL?)
  - Private “modules”

Documentation

- **Website**
  - Developer documentation
  - User documentation
  - C++ API (doxygen)
- **Code snippet / commented examples**
- **Content**
  - Code architecture
    - → developer
  - Main data structures
    - → developer
  - Main “services” usage (Mesh, fields,…)
    - → developer, user
  - End user functionalities (models, time integrators, non-linear solvers, …)
    - → user
  - Theory
    - → user

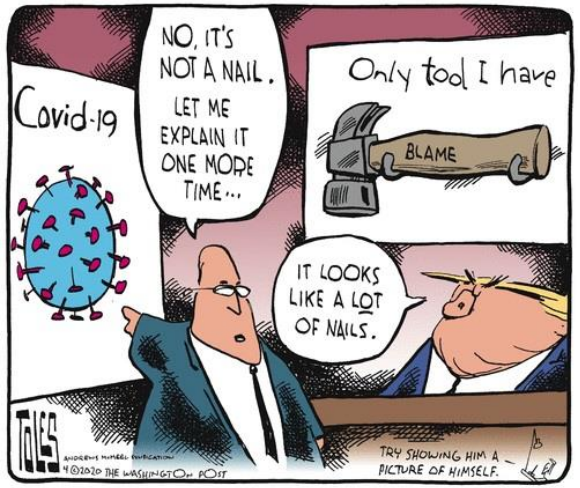




The implicit-explicit unification

Pros

- **Cheaper to develop and maintain**
  - A lot of common features to share, develop and maintain just once
- **Unification of the explicit and implicit developers communities**
  - Clearly separated in the framework of the legacy codes
- **Only one software to master for the end user**
- **Breaks free from the law of the hammer**
  - Abraham Maslow, 1966, "If the only tool you have is a hammer, it is tempting to treat everything as if it were a nail.“
  - For both the end-user …
    - Run better simulation with algorithms more adapted to the considered physical phenomena
  - … and the numerical engineer/scientist
    - Develop better algorithms because the repository code no longer imposes some constraints
    - Enable intra code multiphysics coupling: much more possibilities than with coupling between different codes



Cons

- **Impact on performance ?**
  - What distinguish an “implicit” code from an “explicit” one ?
    - matrices
      - “implicit” requires (sparse) matrices
      - “explicit” lumps mass matrices so that it does not need matrices data structures
    - Non-linearities
      - “implicit” has algorithms to solve non-linear sets of equation over a time/loading step, unconditional stability but lack of robustness
      - “explicit” does not, great robustness, conditional stability (cfl)
    - Cost centers
      - “implicit”: linear system solving
      - “explicit”: assembling of vectors (finite element kinematics, behavior laws, …)
  - No apriori tradeoffs
  - “Implicit-explicit” software structure: implicit code assembling fast
  - Manta treatment of explicit problems: diagonal linear system, specific resolution methods
- **Much less impact on performances than the high level of genericity required for a generalist code targeting industrial applications…**
  - Multi-pde, multi-”zones”, unstructured meshes, …
- **… and “automatic parallelism”**

Development workflow

Collaborative workflow

- **Programming languages**
  - Main language: C++20
  - High level API: python (to come…)
  - Behavior laws: mfront
  - Documentation : reStructuredText

reStructuredText

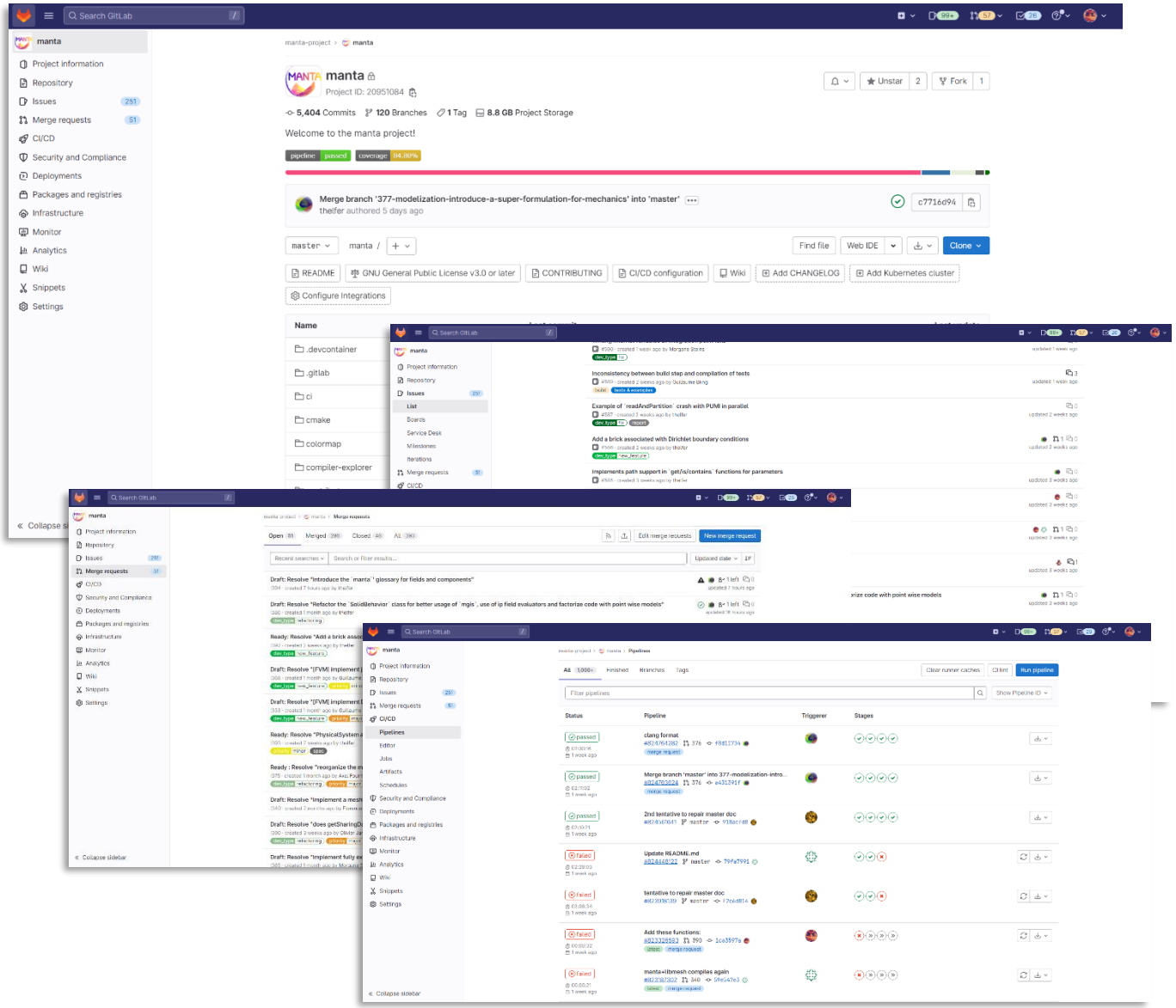
- **Targeted systems**
  - GNU/Linux

  - Laptop → distributed systems with high performance network

- **Very standard workflow and tools**
  - Git: version control

  - Gitlab:
    - Bug/dev tracker, merge requests, labels
    - Code review
    - Tests management (non-regression, V&V)
    - Publish the Documentation
    - Distribute the code

  - CEA's Talkspirit:
    - Discussion forum about MANTA
    - Host shared document about MANTA

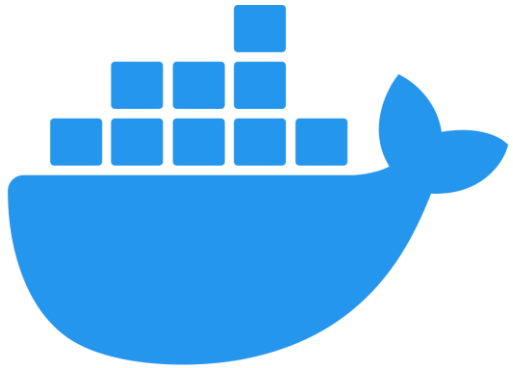


Tests

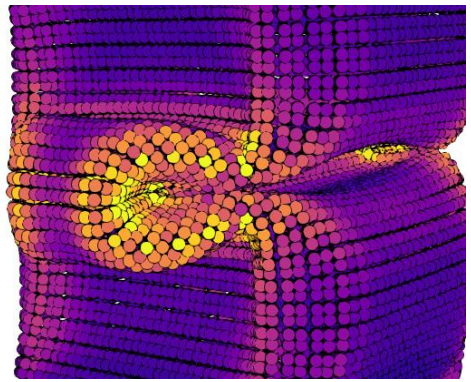
- **Several kind of tests**
  - Non-regression
    - “Integration” (test a single “functionality”)
    - End to end
  - Verification: end to end
  - Validation (not yet …)
- **Every test must be ok to merge a development branch into the master branch in several configurations**
  - Linux distributions
    - Centos 7.9
    - Debian 11
    - Fedora 36
    - Ubuntu 22
  - Build configuration
    - Compiler suite: gcc/clang
    - Optimization level: debug/release
    - Mesh backend: MOAB/PUMI
- **Code snippet in the documentation**
- **Code coverage (non-blocking)**

Quality insurance

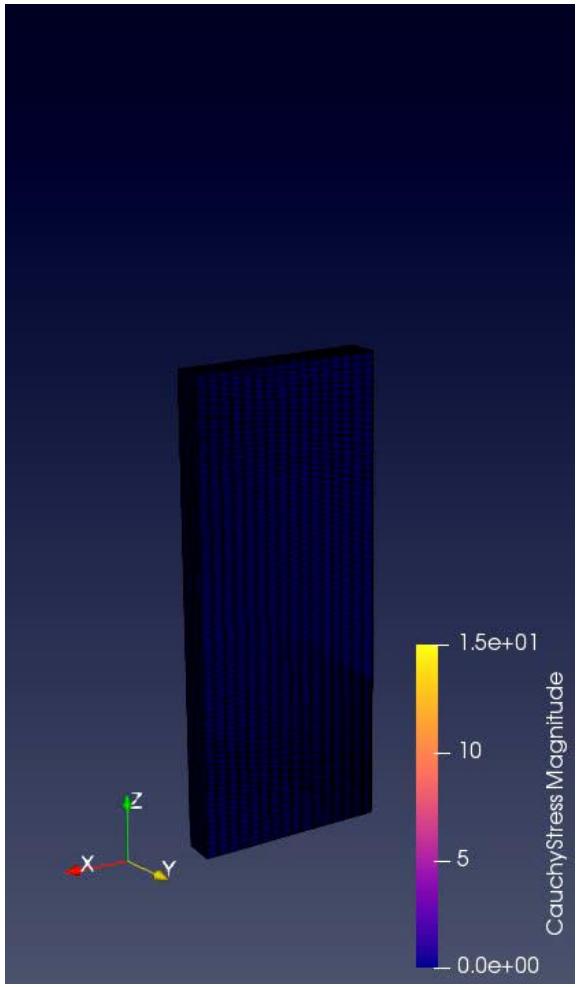
- **Everything is in the repository, for each commit**
  - How to build, with which compiler options: cmake, gitlab.ci
  - What are the 3rd parties and their versions (and recursively): spack configuration file
  - On which tests configurations it has been tested: gitlab.yml
  - How the linux distributions used for testing has been set up: scripts and configuration files to generate and configure docker images
  - The .front files to generate the behaviors
- **We should be able to re-build and re-test every commit in the same conditions as it has been validated**
- **Gitlab**
  - Every modification of the source code is traced in an issue/MR, and reviewed
  - Every commit is related to a MR



A few illustrations



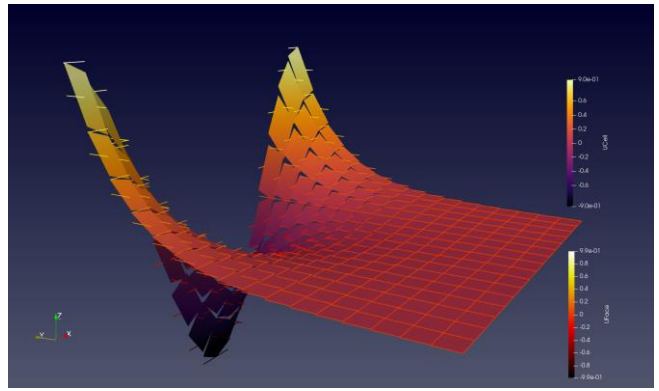
- Dynamic buckling of a beam under impact
- Shell elements



- Rupture of an elastomer
- Quadratic elements
- Transition implicite/explicite

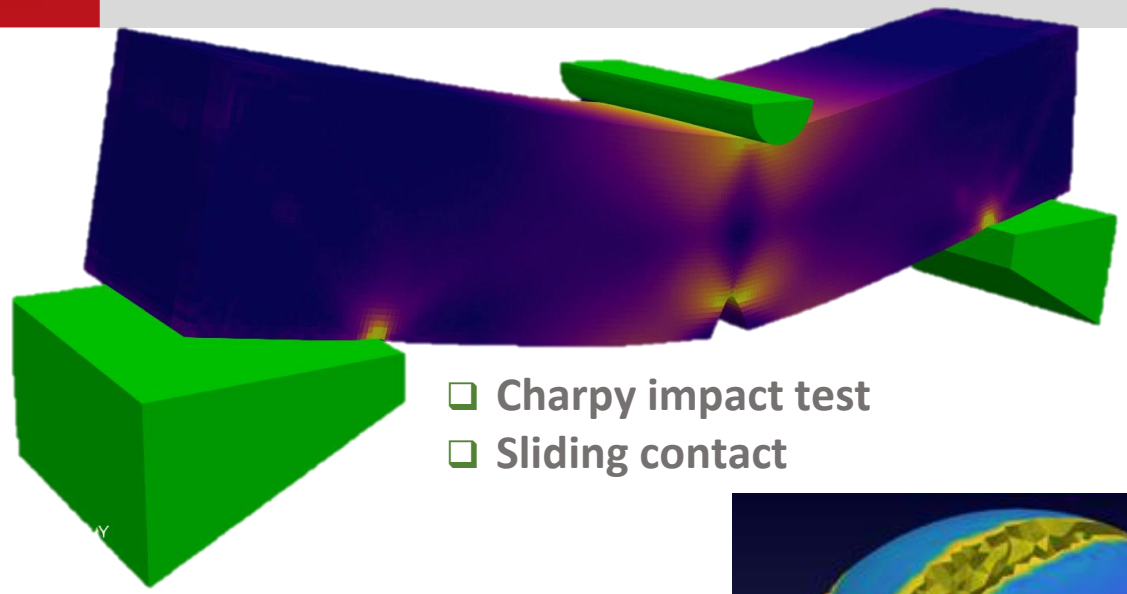
Some features:

- Linear and quadratic finite elements
- Solid and structural elements (shell, beams)
- Large strain
- Complex behaviors (mfront)
- Implicit/Explicit problems
- Sliding contact
- Complex boundary conditions
- Phasefield (explicit/implicit)

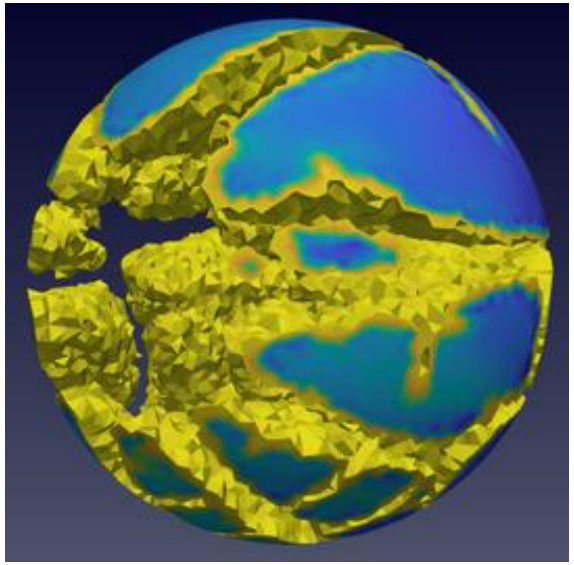


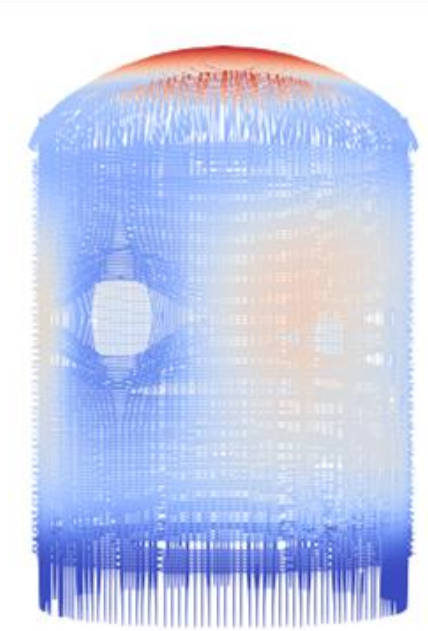
- Problème de Laplacien
- Méthode HHO

Finite elements, structural mechanics, HHO

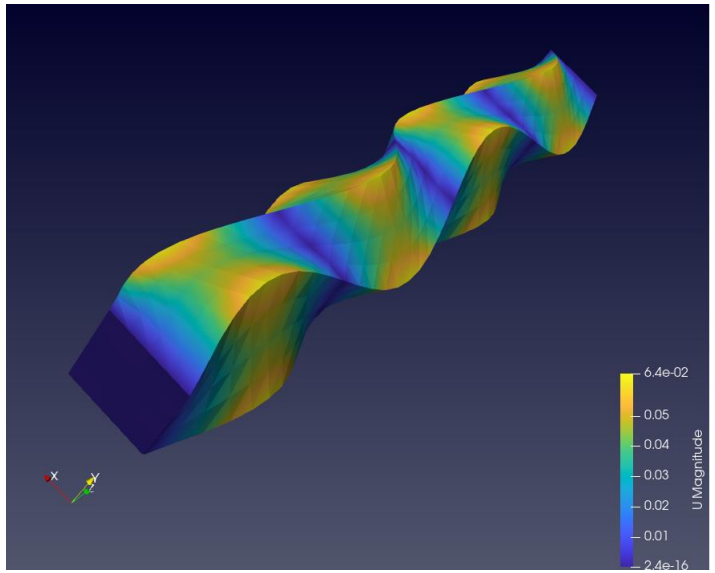


- Charpy impact test
- Sliding contact





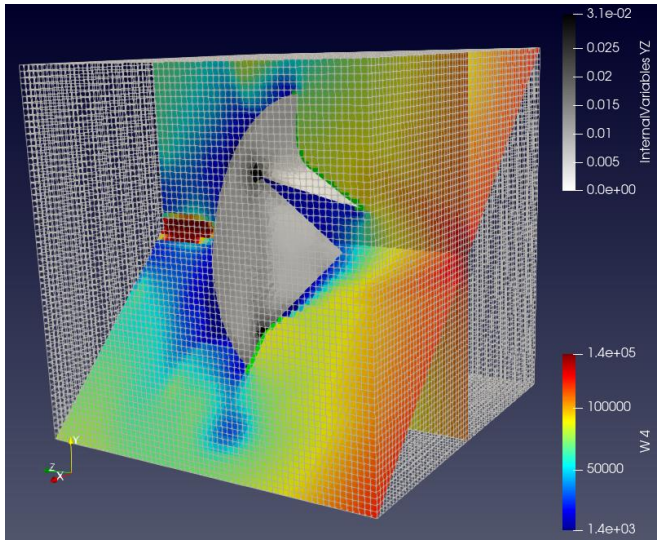
- Pressurized build
- Reinforced concrete



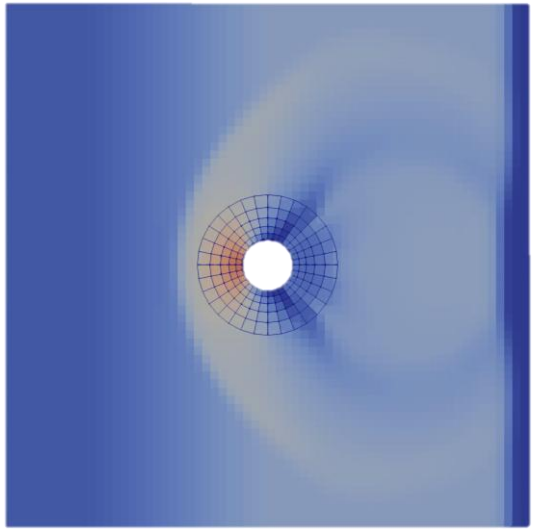
- Modal solver

- Impact of a marble on a plane
- Phasefield

Finite-volumes, compressible fluid mechanics, FSI



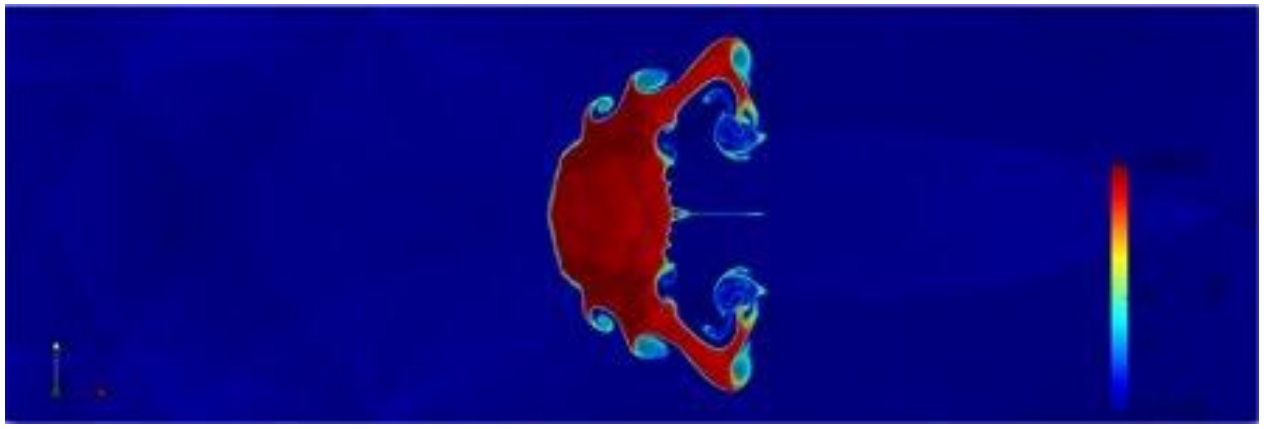
- Explosion near a metallic circular plate
- Immersed boundaries FSI
- MBM



- Introduction of a cylinder into a flow through a local model
- Chimera method

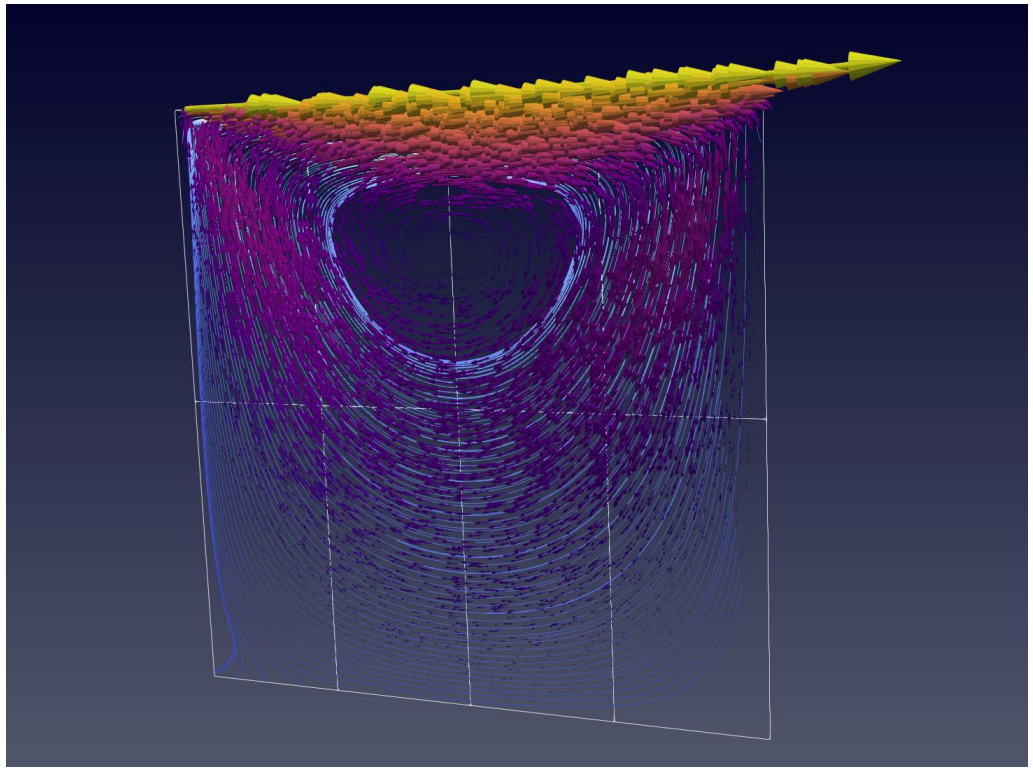
Some features:

- Order 2 finite volumes
- FSI immersed boudaries (Mediating Body M ethod)
- Implicit/explicit temporal integration
- Euler equations
- Multi-component flows (5-equations model)
- Chimera method



- Bubble-shock interaction
- Multi-component flow

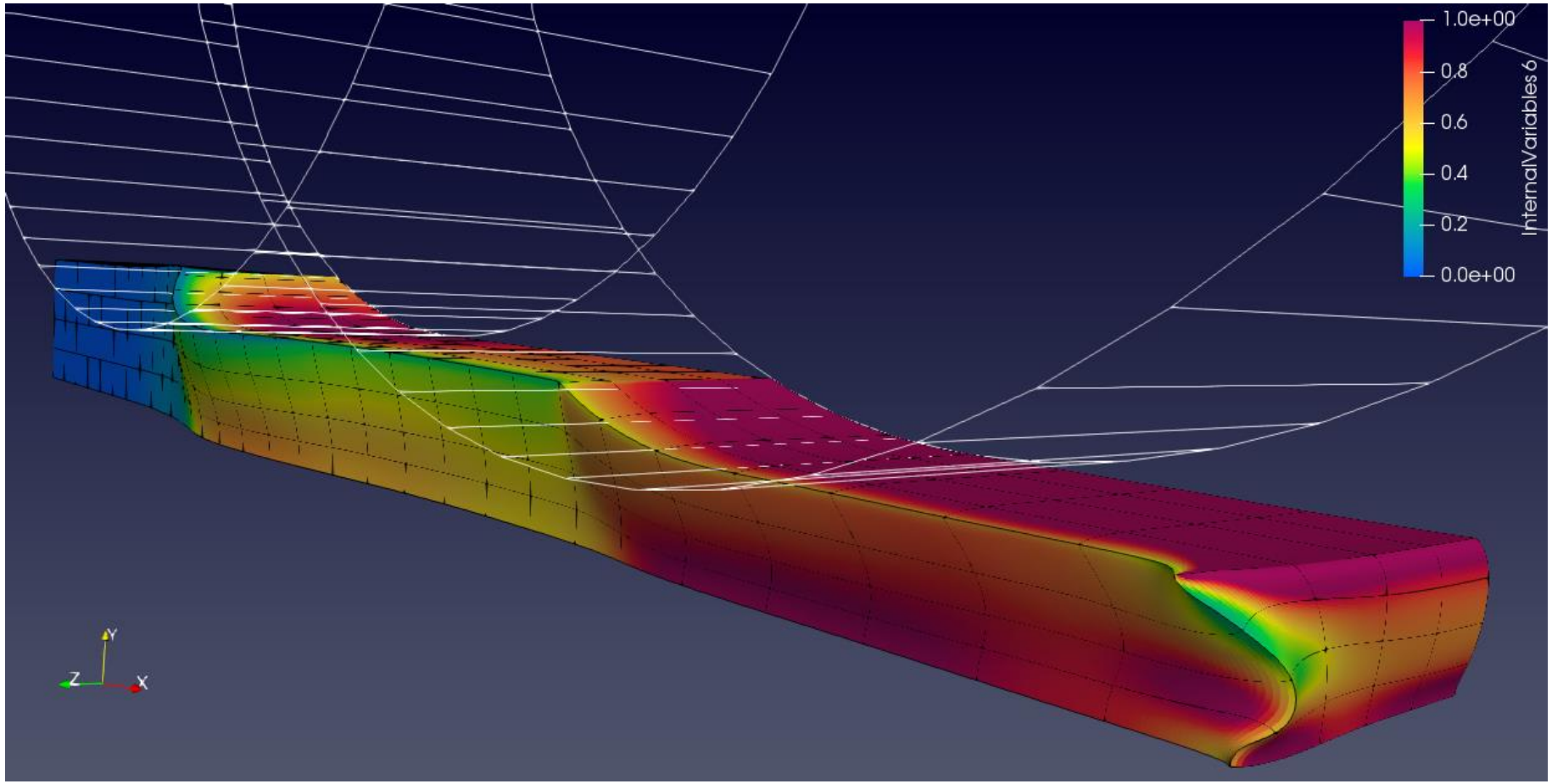
Finite-elements, incompressible fluid mechanics



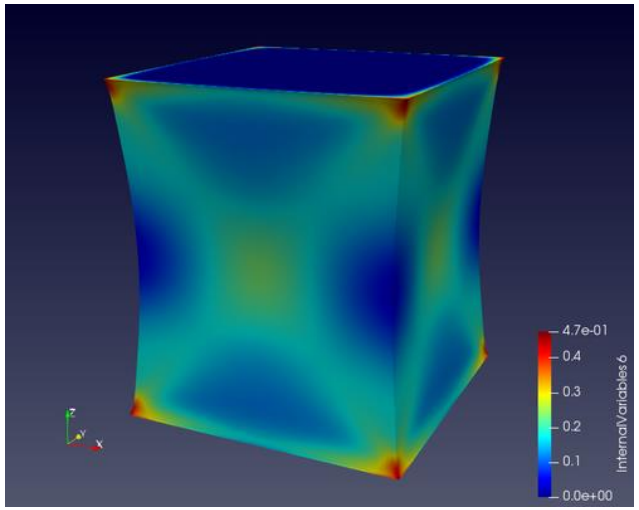
- Classical test case: lid driven viscous fluid in cavity
- Low Reynolds
- Stationnary problem

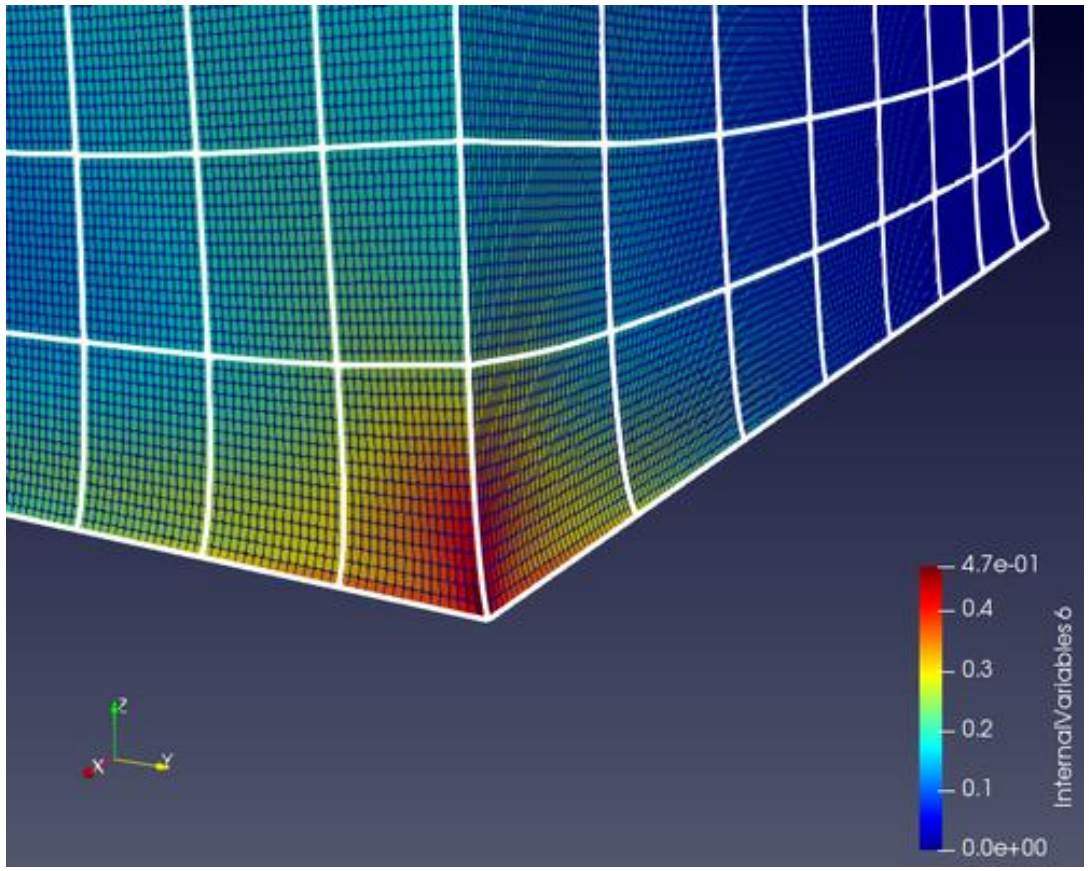
Some features:

- Stokes problem
- Taylor-hood elements

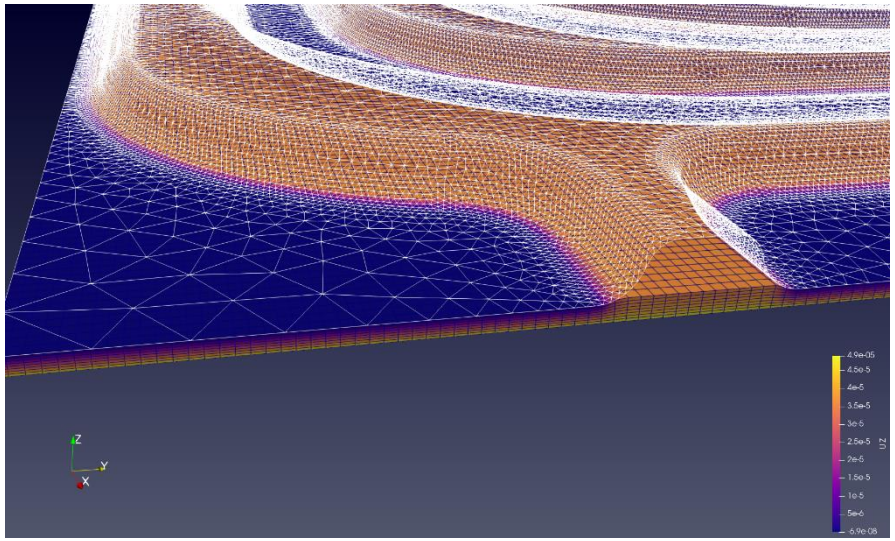


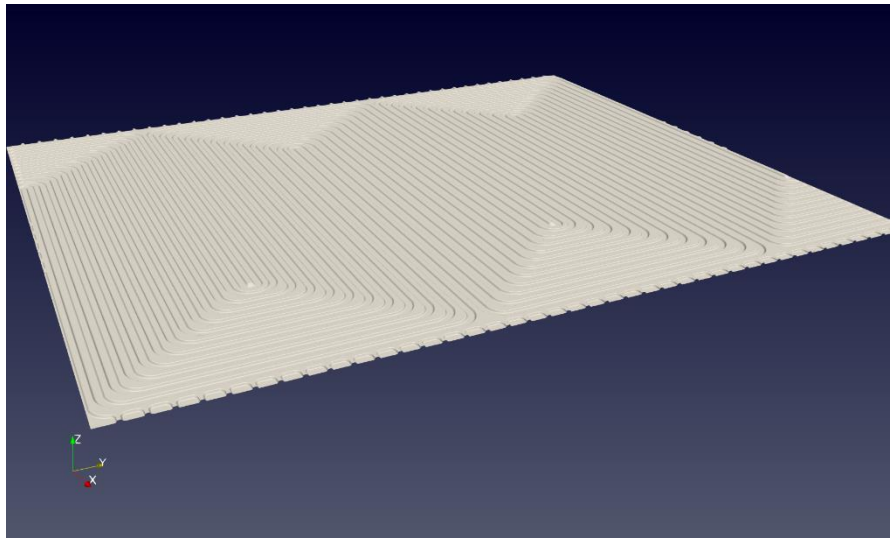
- Rolling of a steel billet
- Explicit calculation
- Sliding contact between the mills and the billet
- $24x10^6$ dofs
- 512 MPI subdomains
- ~200000 time steps

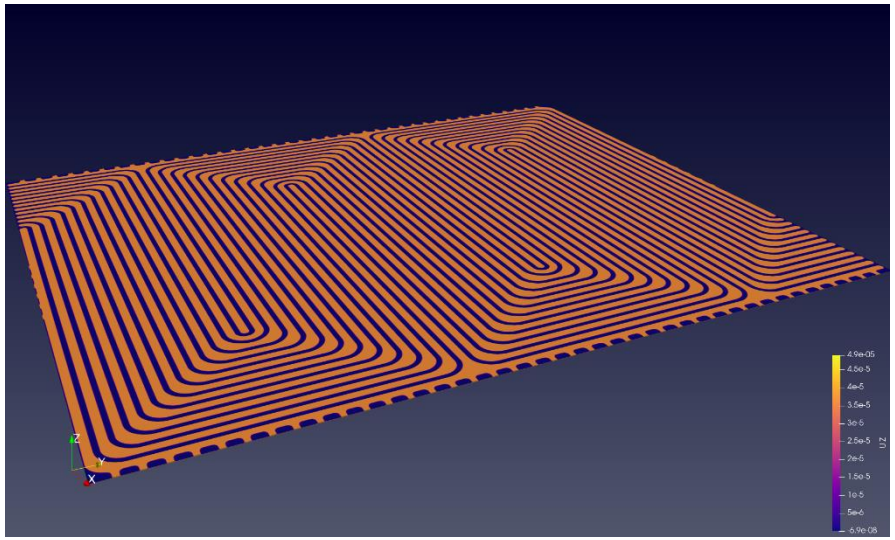


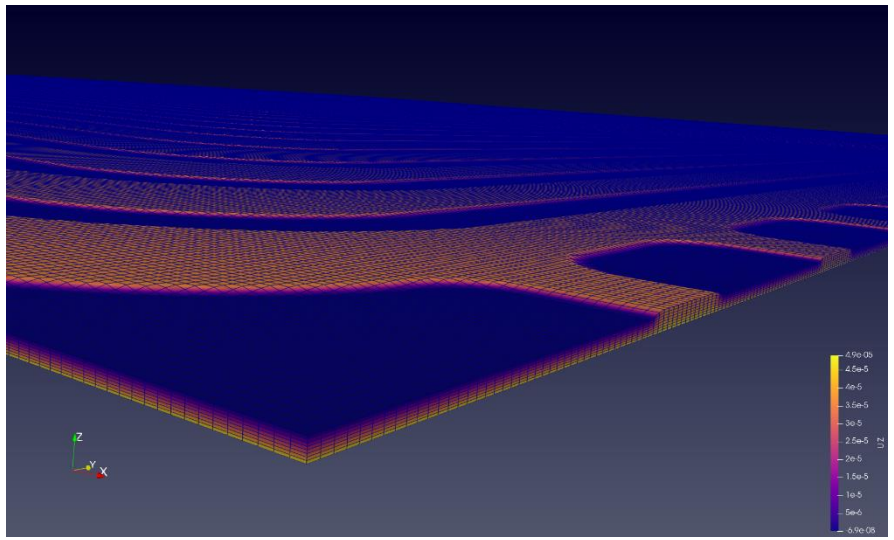


- Traction on a steel cube
- Implicit calculation
- 16% average strain (~50% locally)
- 51×10^6 dofs
- 2048 MPI subdomains
- 40 loading steps



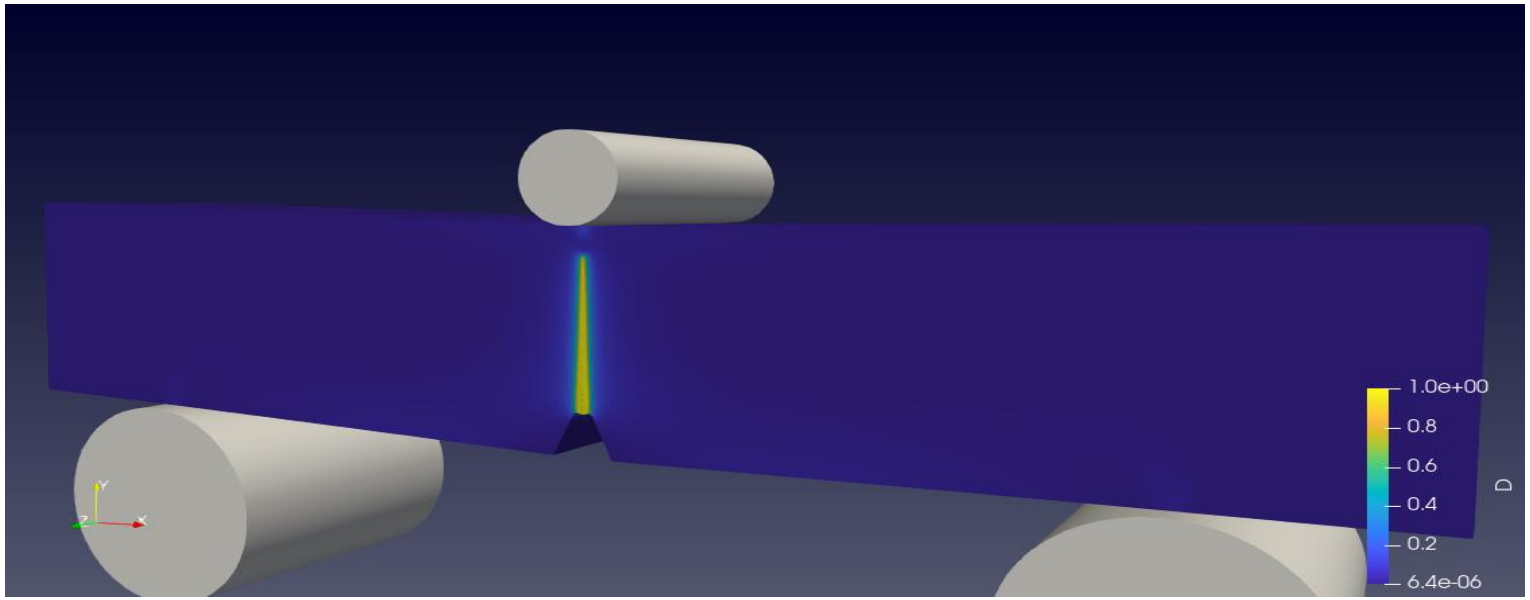




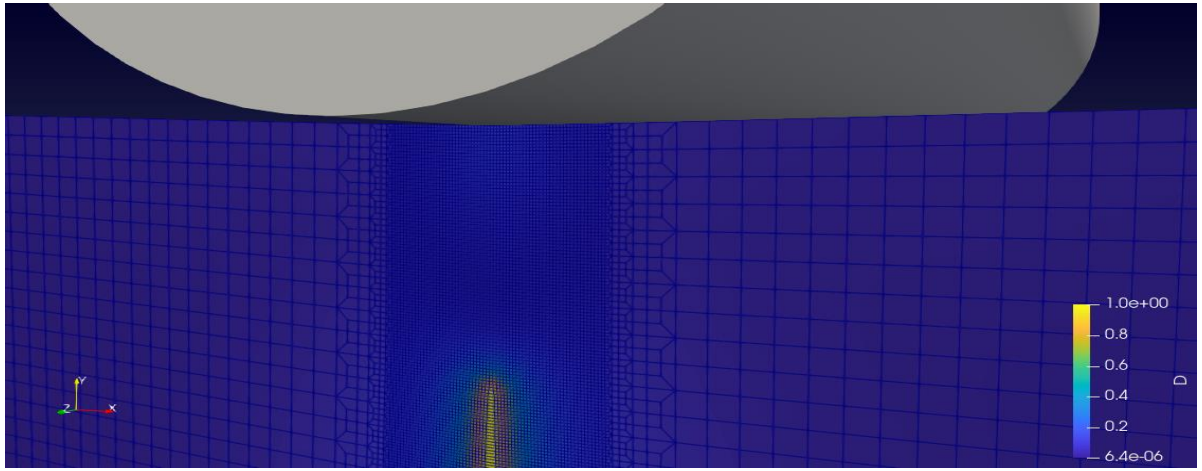


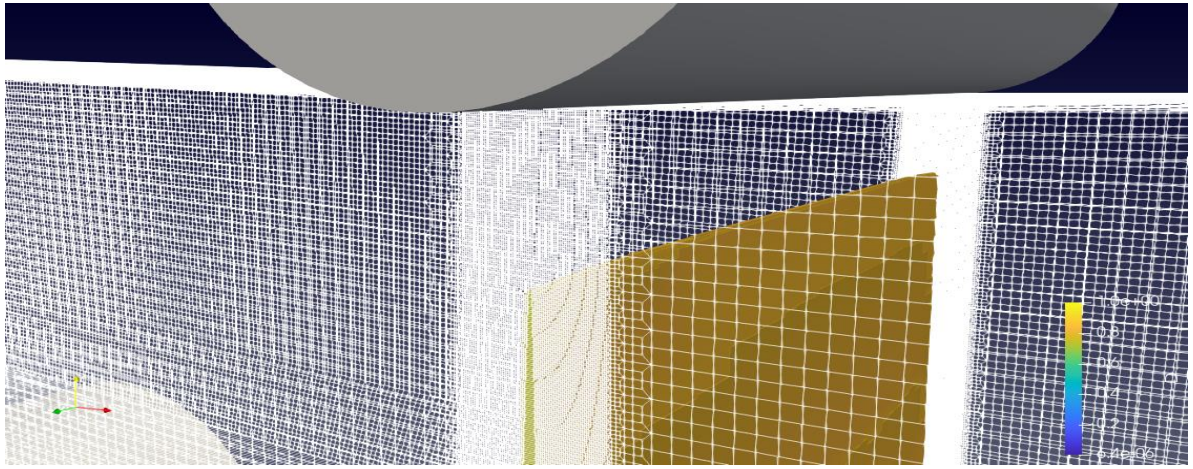
- Plate metal forming for fuel cell
- Explicit calculation
- Sliding Contact
- 84×10^6 dofs, 24×10^6 cells
- « Punch » surface : 12×10^6 triangles
- 1000 MPI subdomains

Galerie: HPC

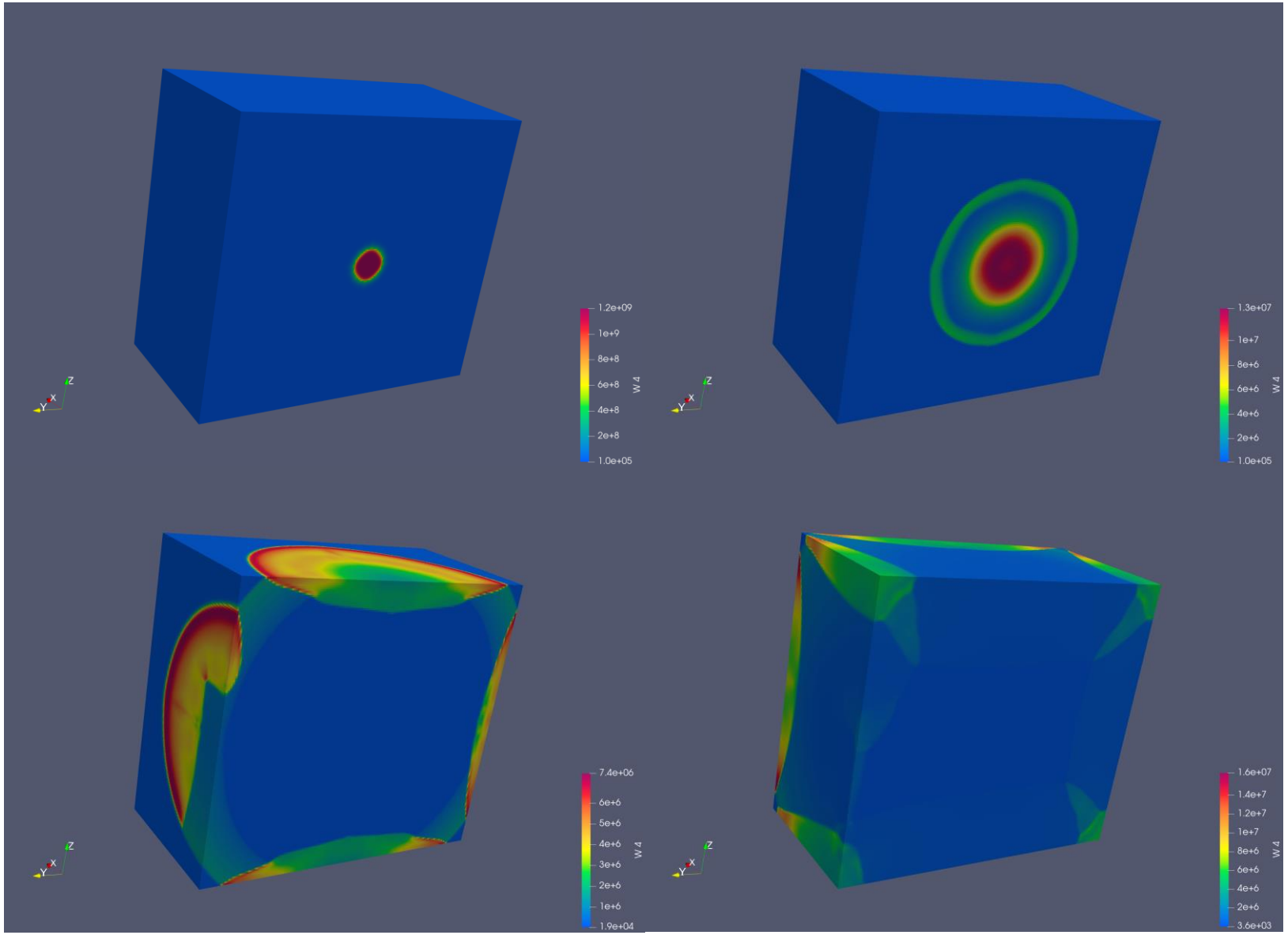


- Charpy dynamic impact test
- Explicit calculation
- Slidding contact
- Phasefield
- 67x10e^6 cells
- 10 000 time steps
- 8192 MPI subdomains





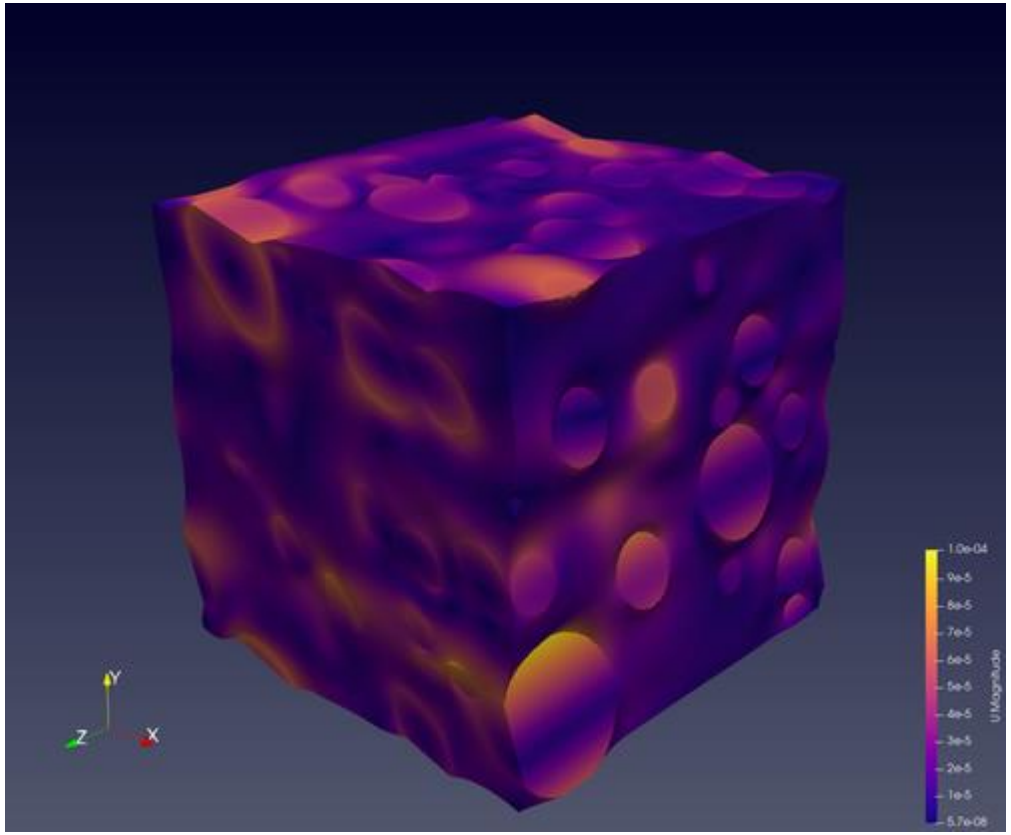
Galerie: HPC



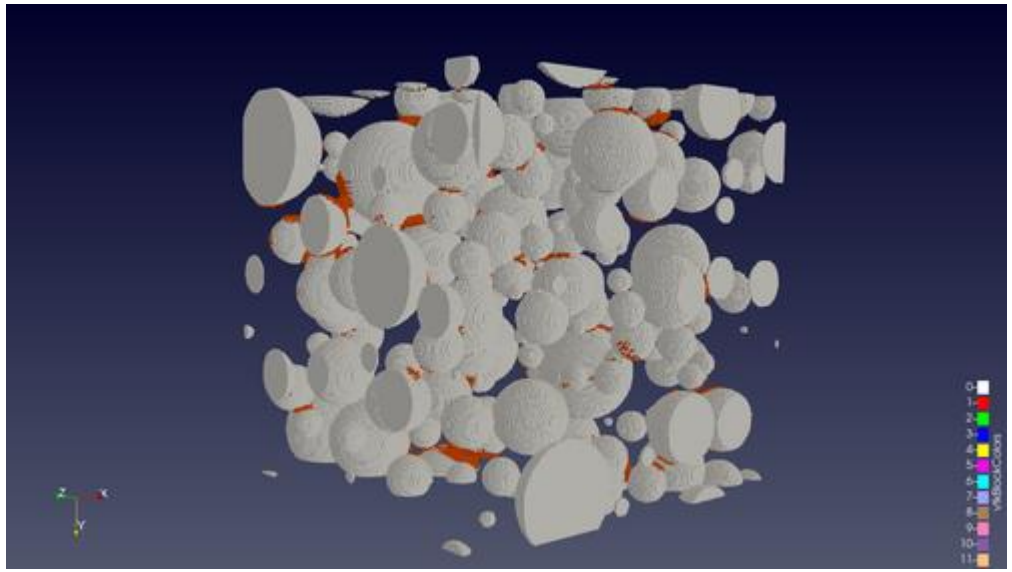
- “explosion” simulation
- Order 2 finite volumes
- 160x10e6 cells
- 8192 MPI subdomains
- Explicit calculation

Visu: 300k cells

Galerie: HPC



- REV simulation
- 2-phase: elastic inclusions and elasto-plastic matrix
- Contrast: 100
- Periodic boundary conditions
- 400x10e6 dofs
- 16384 MPI subdomains
- Implicit calculation



Ongoing works

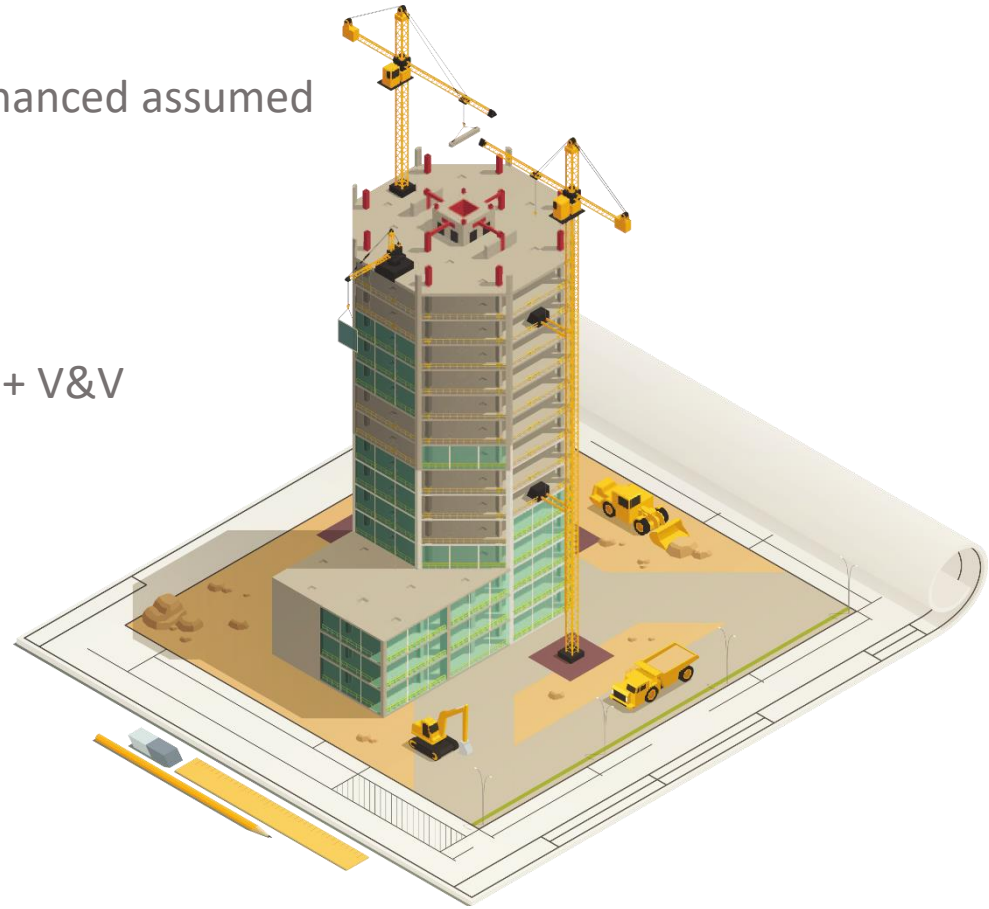
Ongoing and future works

Generic works

- AMR
- // IO with MED
- Save/restart
- Robust non-linear solver
- Efficient solvers and preconditioners
- Performance portability (GPUs)
- APIs
  - C++ “easy”
  - Python
  - Plugins system
- Documentation

And specific functionalities related to target applications

- Contact mechanics
- Anti-hourglass, enhanced assumed strains
- Erosion
- And much more … + V&V



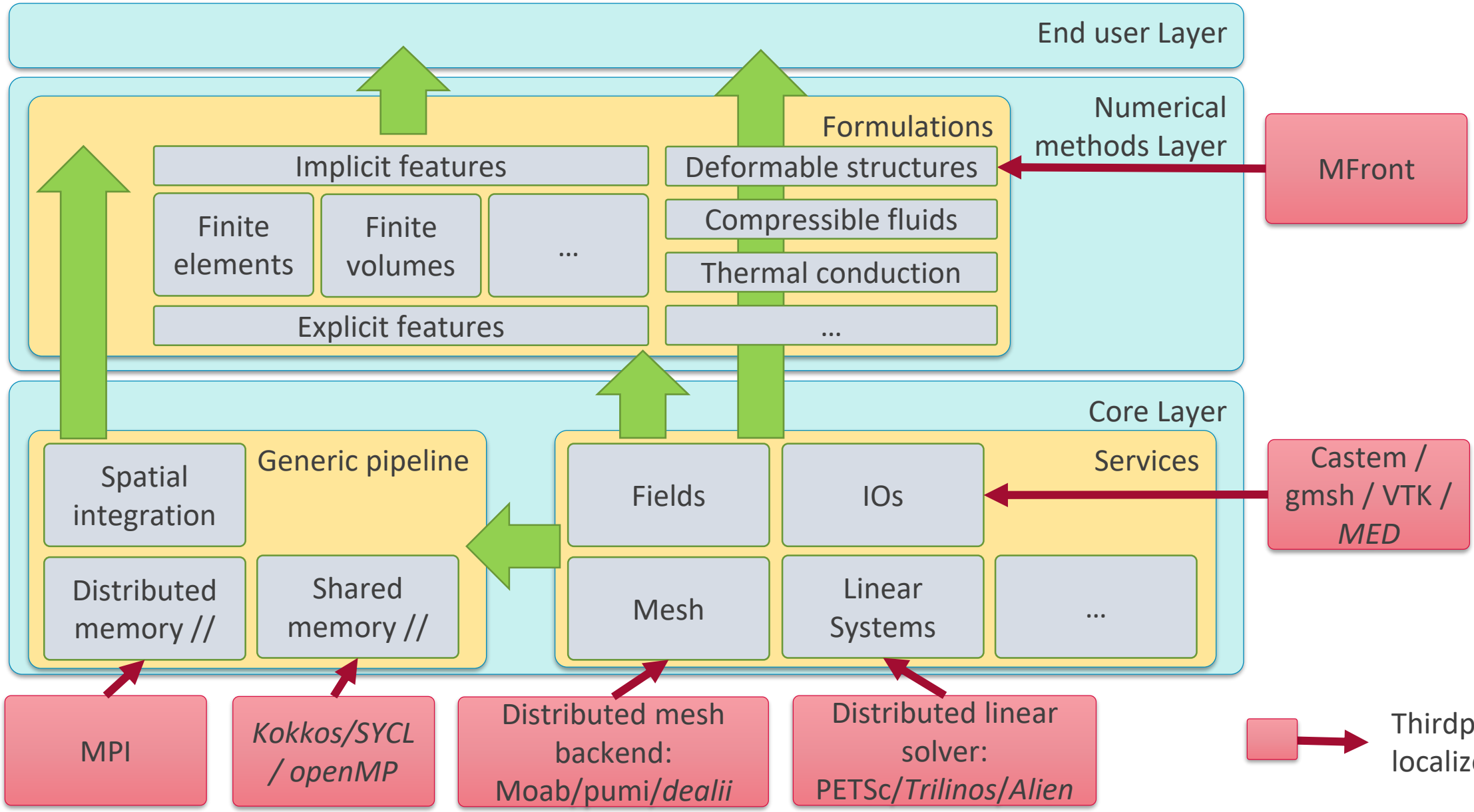
DE LA RECHERCHE À L'INDUSTRIE

Thanks for your attention!

Some questions?

Software architecture overview

Overview


End user Layer
Numerical methods Layer
Formulations
Implicit features
Finite elements
Finite volumes
...
Explicit features
Deformable structures
Compressible fluids
Thermal conduction
...
MFront
Core Layer
Generic pipeline
Spatial integration
Distributed memory //
Shared memory //
Services
Fields
IOs
Mesh
Linear Systems
...
Castem / gmsh / VTK / MED
MPI
Kokkos/SYCL / openMP
Distributed mesh backend: Moab/pumi/dealii
Distributed linear solver: PETSc/Trilinos/Alien
Thirdparty with localized interface

Generic pipeline: purpose

Purpose

- Assemble distributed linear systems resulting from spatial integration on meshes
- Attach “constraints” to linear systems
- Solve the (saddle point problems) linear systems
- Support all the parallelism

$$\begin{bmatrix} A & C0^t & C1^t & \cdots \\ C0 & 0 & 0 & \cdots \\ C1 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \mathrm{X} \\ \lambda 0 \\ \lambda 1 \\ \vdots \end{bmatrix} = \begin{bmatrix} B \\ D0 \\ D1 \\ \vdots \end{bmatrix}$$

Spatial integration

- Split global integral over mesh entities: $\boldsymbol{M} = \sum_i \mathcal{A}_i \int_{E_i} \boldsymbol{m}(\underline{x})\, dx$
- Use finite-element mapping with reference element to integrate using standard quadrature formulae:

$$\boldsymbol{M} = \sum_i \mathcal{A}_i \sum_j w_j \boldsymbol{m}(\underline{\xi}_j) |det(\underline{\phi}_i(\underline{\xi}_j))| \text{ , where } (\underline{x} \in E_i) = \underline{\phi}_i(\xi)$$

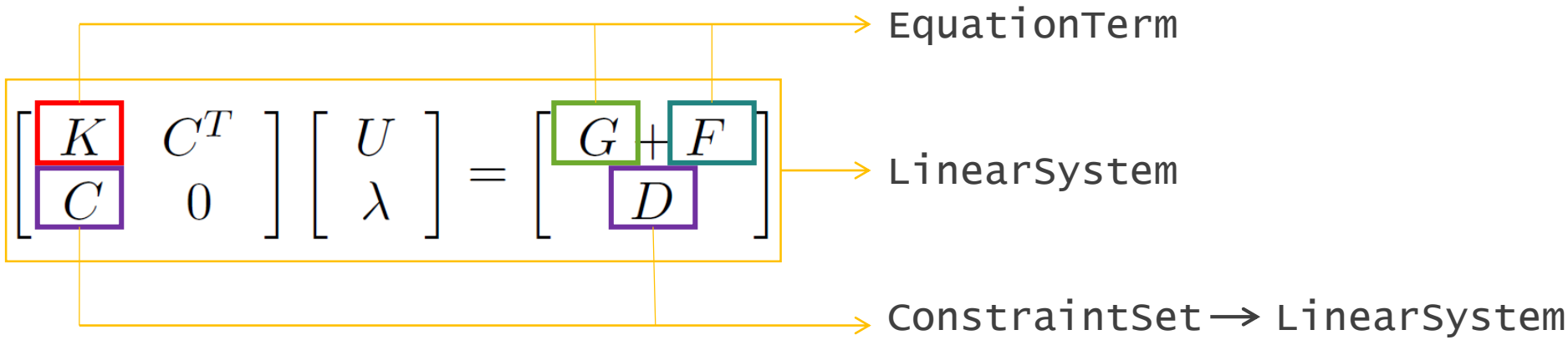
- Main entry points:
  - `Integrand::addOn` → $w_j \boldsymbol{m}(\underline{\xi}_j) |det(\underline{\phi}_i(\underline{\xi}_j))|$
  - `Assembler::assemble` → $\mathcal{A}_i$

Generic pipeline: assembling a linear system

- `EquationTerm`
  - $AX = B \rightarrow A = A_O + A_1 + \cdots, \mathrm{B} = \mathrm{B}_0 + \mathrm{B}_1 + \cdots$
  - Each `EquationTerm` instance "contains" an `Integrand` and an `Assembler`
- `Formulation`
  - End-user interface
  - Contains a set of `EquationTerm`
  - Defined on a `MeshSet` (a "physical zone")
  - Can provide specific functions to the end-user
  - Meant to be "attached" to a linear system to add its `EquationTerm` computations when `LinearSystem::assemble` is called.

Example of the scalar laplacian

$$\begin{cases} \int_\Omega \nabla^s u : \nabla^s v \, dx = \int_\Omega g \cdot v \, dx + \int_{\partial\Omega_N} f \cdot v \, dx \\ u = u_0 \text{ on } \partial\Omega_D \end{cases}$$



$$\begin{bmatrix} K & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} U \\ \lambda \end{bmatrix} = \begin{bmatrix} G + F \\ D \end{bmatrix}$$

$$K = \sum_{\{C_i\}} \mathcal{A}_i \left(\sum_{\{IP\}_{C_i}} B^T B \, det(J) \, w \right)$$

Assembler

Integrand

Generic pipeline: solving/using a linear system

Lagrange multipliers elimination

$$\begin{bmatrix} A & C0^t & C1^t & \cdots \\ C0 & 0 & 0 & \cdots \\ C1 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \mathrm{X} \\ \lambda 0 \\ \lambda 1 \\ \vdots \end{bmatrix} = \begin{bmatrix} B \\ D0 \\ D1 \\ \vdots \end{bmatrix}$$

- Several methods available
  - Inexact: penalty method
  - Exact: Ainsworth method, master/slave method
- The application of elimination methods is `ConstraintSet`-wise
- The non-eliminated `ConstraintSet` are gathered. The saddle-point system is treated by PETSc.
  - By default, as a “Nested” system (“Fieldsplit” preconditionners)
    - Schur elimination, Golub-Kahan method, …
  - Can be turned by manta to a “monolithic” system (mainly for debugging purposes)
  - Pass solver options using PETSc’s dynamic options system: CLI or file

Encapsulation

- The end-user never handles the internal distributed matrices/vectors
- Get/set data using manta’s fields
- `LinearSystem::updateField`, `LinearSystem::applyLSH`, `LinearSystem::getRHS`, `LinearSystem::setRHS`, …

Generic pipeline: parallelism

MPI only at this time

- Decomposition of the global mesh into subdomains: each MPI process works out and stores only its subdomain (1 subdomain per MPI process)
  - "Almost (ghosts) Total" distribution of data

Current work for performance portability

- Hybrid MPI+CPU-threads is not a goal in itself

"Automatic" parallelism

- Generic "pipeline"
  - only loops over local entities for the integration (but the entry points can use ghost cells) → implementing the entry points is the same as in sequential
- Ghosting
  - Each process can replicate any mesh cell owned by another process → ghost cell
  - When imported, a ghost entity carries all the data it is related to (*e.g.* `MeshSet` belongings), and recursively for its lower dimensional entities (may induce an excess of communication volume)
  - A ghost entity (as a local one) should be the same as in sequential
  - Functions to synchronize field values on ghost entities

Main “services”

- Mesh
- Data fields
- “Zones” management
- Dense linear algebra
- Linear system, sparse matrix handling
- IOs
- Shape functions (Lagrange, serendipity, monomial basis)
- Configurations (initial, current, last) management for Lagrangian/ALE approaches
- Computational geometry
- Generic algorithms (Newton-Raphson, …)

Ongoing and future works

- Non conforming AMR
- Performance portability
- IO //
- Save/restart, fault tolerance
- Distributed computational geometry
- APIs, documentation, distribution, …