Parallel Computations

Timo Heister, Clemson University
heister@clemson.edu

2015-08-05
deal.II workshop 2015

CLEMSON UNIVERSITY

CIG COMPUTATIONAL INFRASTRUCTURE for GEODYNAMICS

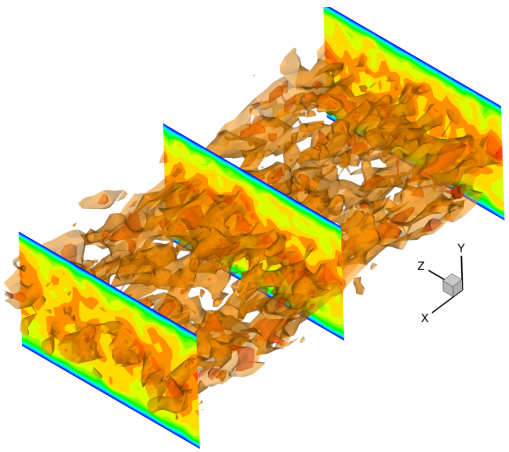
Introduction

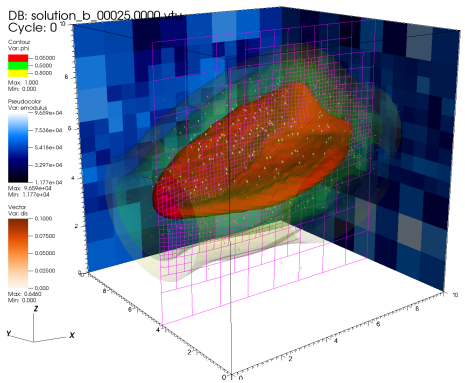
- Parallel computations with deal.II: Introduction
- Applications
- Parallel, adaptive, geometric Multigrid
- Ideas for the future: parallelization

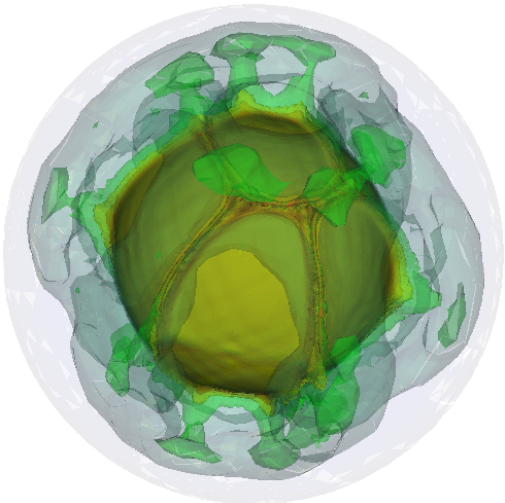
My Research

1. Parallelization for large scale, adaptive computations
2. Flow problems: stabilization, preconditioners
3. Many other applications

IBM Sequoia, 1.5 million cores, source: nextbigfuture.com







Parallel Computing

	Before	**Now (2012)**
Scalability	ok up to 100 cores	16,000+ cores
# unknowns	maybe 10 million	5+ billion

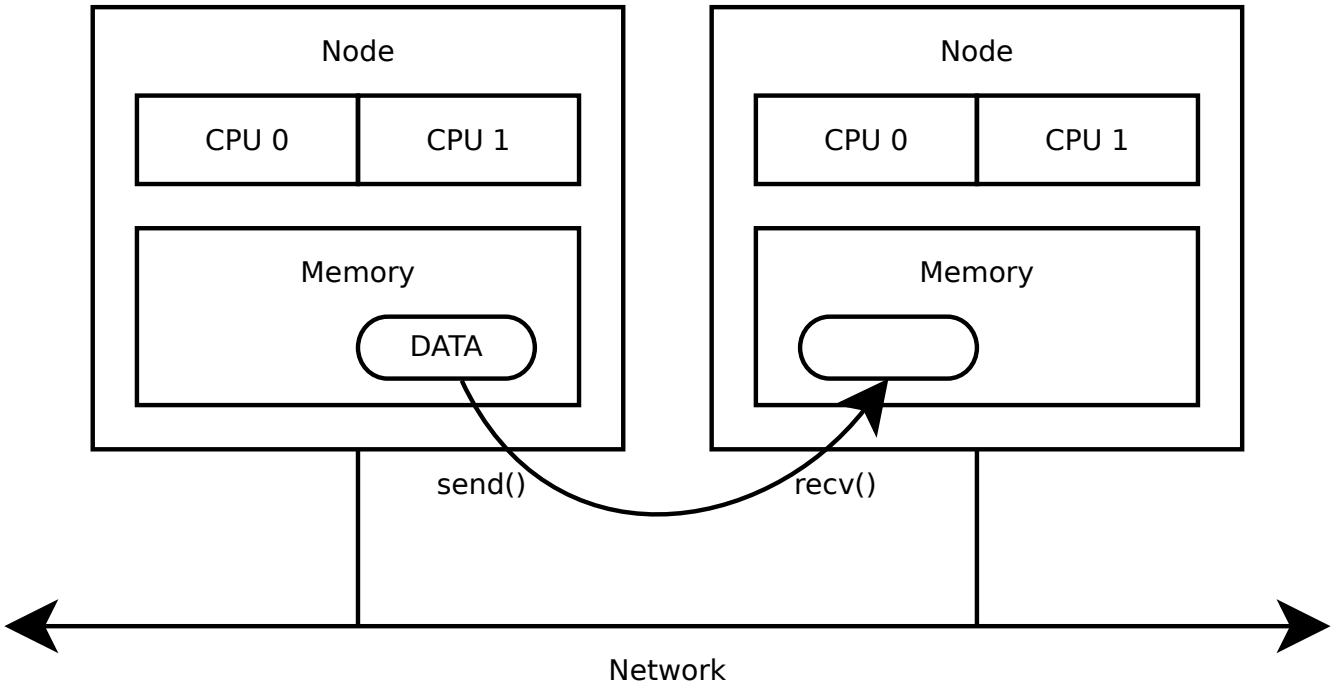
- Ideas:
  - Fully parallel, scalable
  - Keep flexibility(!)
  - Abstraction for the user
  - Reuse existing software
- Available in `deal.II`, but described in a generic way:

Bangerth, Burstedde, Heister, and Kronbichler.
Algorithms and Data Structures for Massively Parallel Generic Finite Element Codes.
ACM Trans. Math. Softw., 38(2), 2011.

Parallel Computing Model

- System:
  nodes connected via fast network
- Model: MPI
- we here ignore:
  multithreading and vectorization

IBM Sequoia, 1.5 million cores,
source: nextbigfuture.com



Parallel Computations: How To? Why?

- Required: split up the work!
- Goal: get solutions faster, allow larger problems
- Who needs this?
  - 3d computations?
  - $> 500,000$ unknowns?
- From laptop to supercomputer!

Scalability

What is scalability?
(you should know about weak/strong scaling, parallel efficiency, hardware layouts, NUMA, interconnects, . . .)

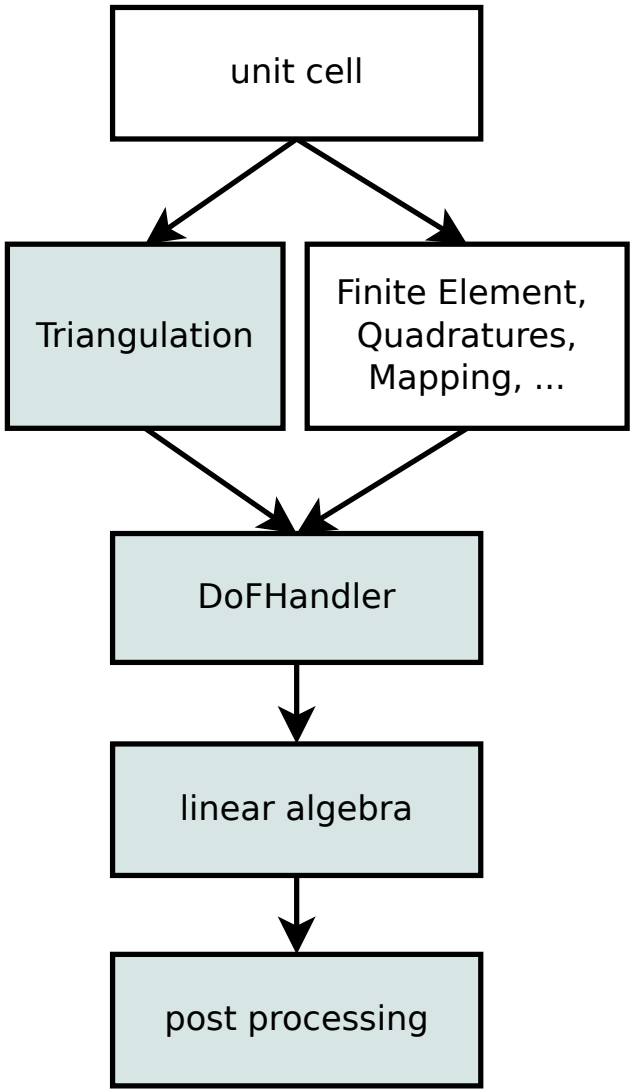
Required for Scalability:

- Distributed data storage everywhere
  $\rightsquigarrow$ need special data structures
- Efficient algorithms
  $\rightsquigarrow$ not depending on total problem size
- "Localize" and "hide" communication
  $\rightsquigarrow$ point-to-point communication, nonblocking sends and receives

Overview of Data Structures and Algorithms

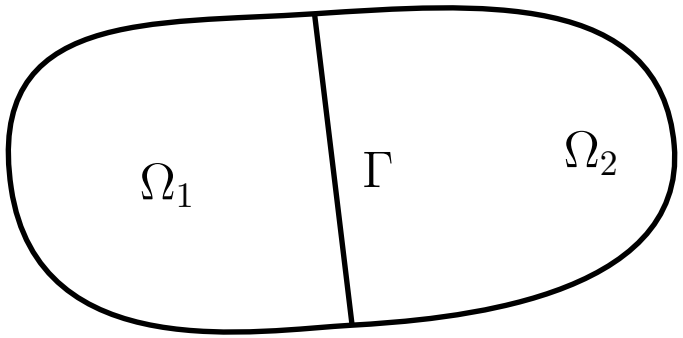
Needs to be parallelized:

1. Triangulation (mesh with associated data)
   — hard: distributed storage, new algorithms
2. DoFHandler (manages degrees of freedom)
   — hard: find global numbering of DoFs
3. Linear Algebra (matrices, vectors, solvers)
   — use existing library
4. Postprocessing (error estimation, solution transfer, output, . . .)
   — do work on local mesh, communicate



How to do Parallelization?

Option 1: Domain Decomposition

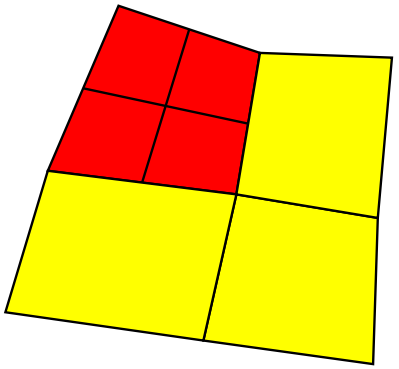


- Split up problem on PDE level
- Solve subproblems independently
- Converges against global solution
- Problems:
  - Boundary conditions are problem dependent:
    $\rightsquigarrow$ sometimes difficult!
    $\rightsquigarrow$ no black box approach!
  - Without coarse grid solver:
    condition number grows with # subdomains
    $\rightsquigarrow$ no linear scaling with number of CPUs!

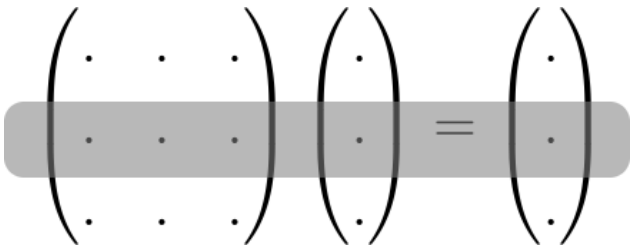
How to do Parallelization?

Option 2: Algebraic Splitting

- Split up mesh between processors:



- Assemble logically global linear system (distributed storage):

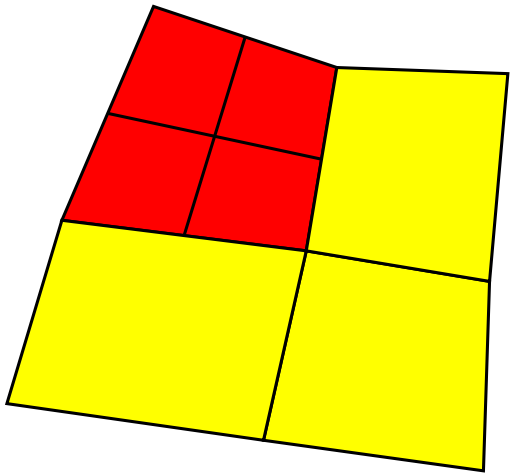


- Solve using iterative linear solvers in parallel
- Advantages:
  - Looks like serial program to the user
  - Linear scaling possible (with good preconditioner)

Partitioning

Optimal partitioning (coloring of cells):

- same size per region
  $\rightsquigarrow$ even distribution of work
- minimize interface between region
  $\rightsquigarrow$ reduce communication



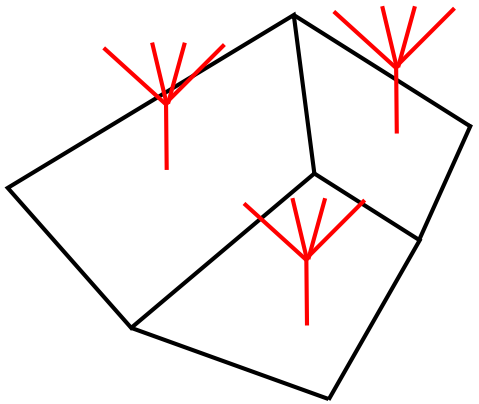
Optimal partitioning is an NP-hard graph partitioning problem.

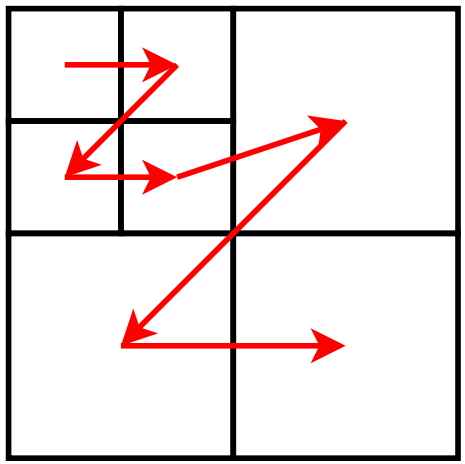
- Typically done: heuristics (existing tools: METIS)
- Problem: worse than linear runtime
- Large graphs: several minutes, memory restrictions

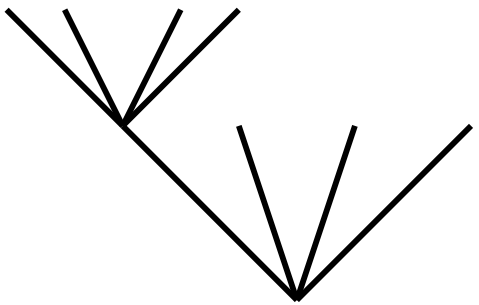
$\rightsquigarrow$ Alternative: avoid graph partitioning

Partitioning using Space-Filling Curves

- *p4est* library: parallel quad-/octrees
- Store refinement flags from a base mesh
- Based on space-filling curves
- Very good scalability



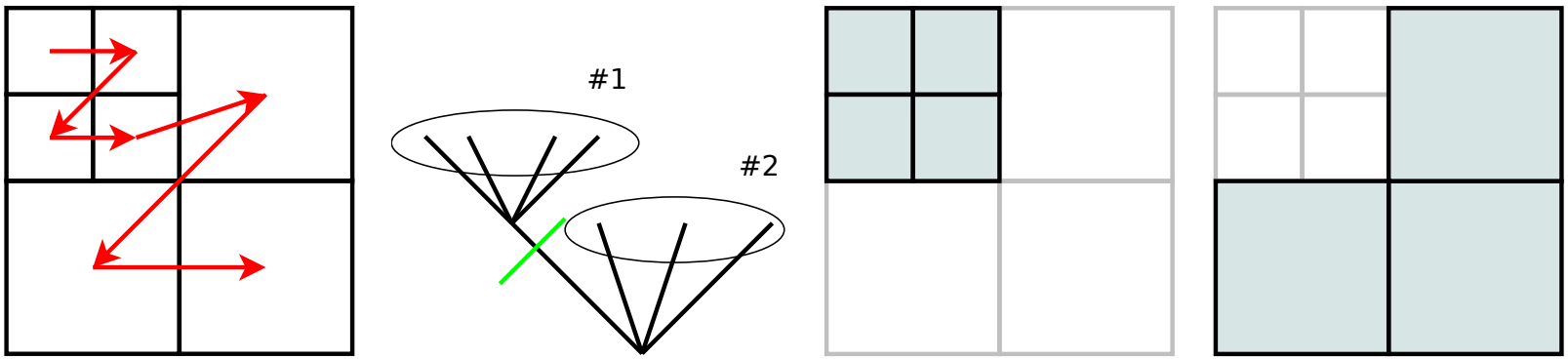




Burstedde, Wilcox, and Ghattas.
p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees.
SIAM J. Sci. Comput., 33 no. 3 (2011), pages 1103-1133.

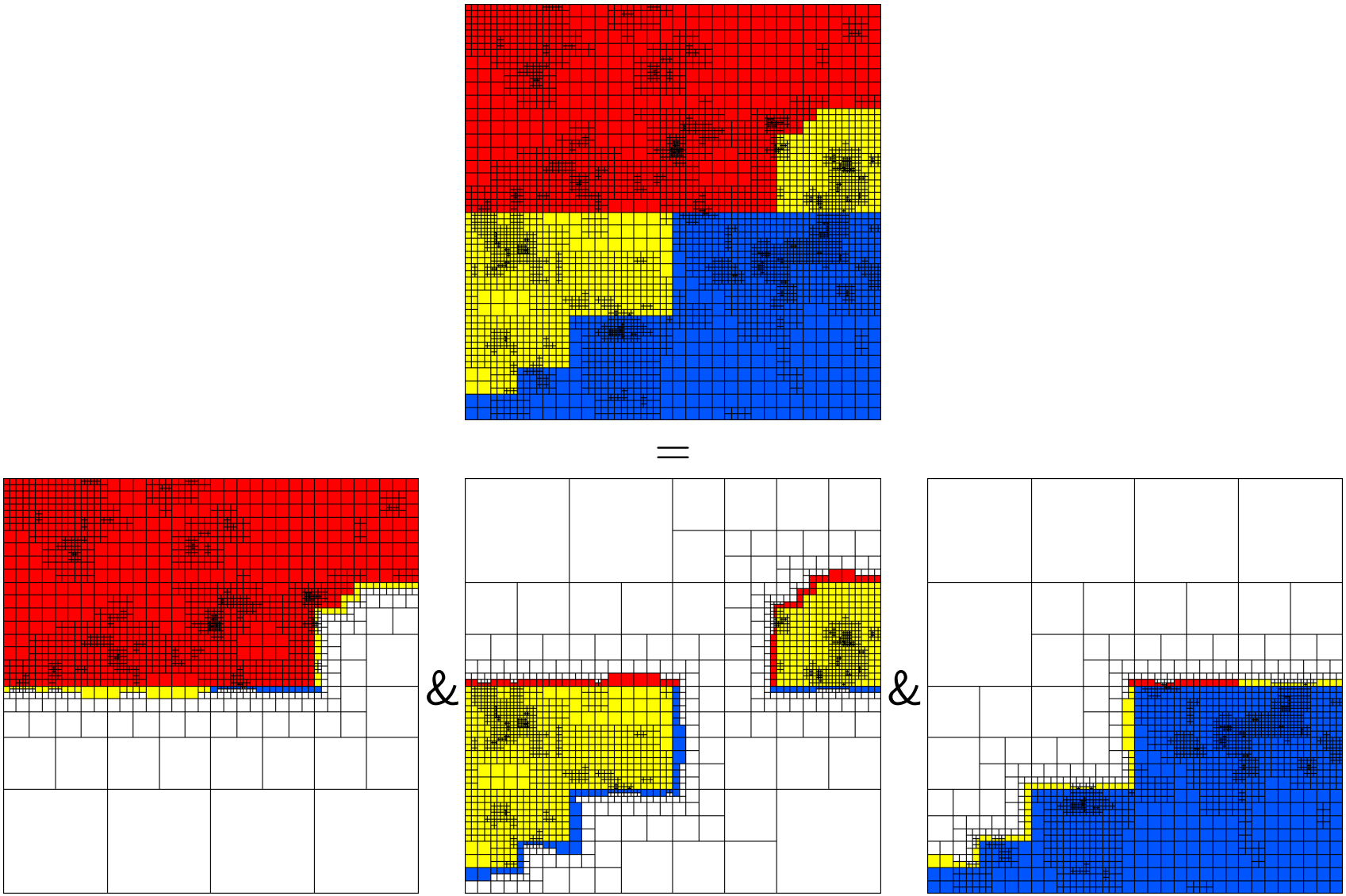
Triangulation

- Partitioning is cheap and simple:



- Then: take *p4est* refinement information
- Recreate rich *deal.II* Triangulation only for local cells (stores coordinates, connectivity, faces, materials, . . .)
- How? recursive queries to *p4est*
- Also create ghost layer (one layer of cells around own ones)

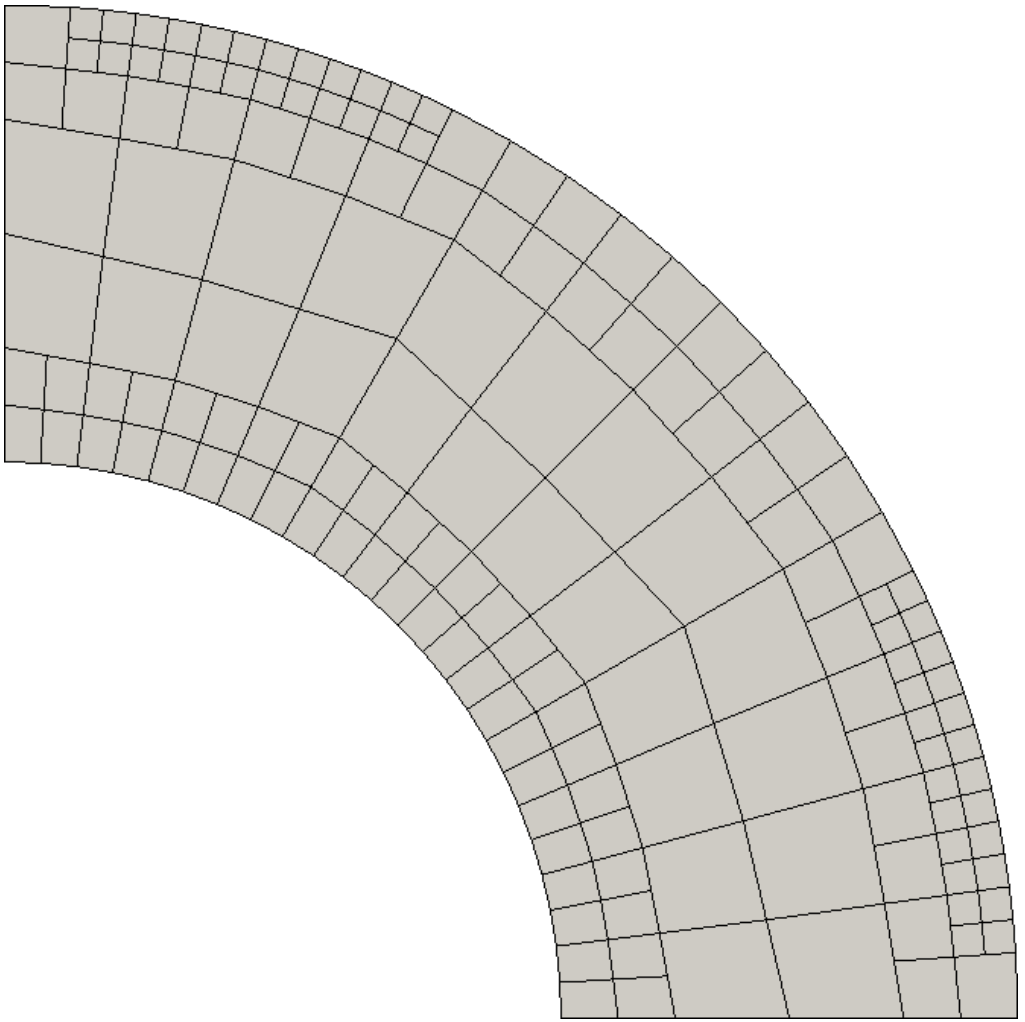
Example: Distributed Mesh Storage



Color: owned by CPU id

Arbitrary Geometry and Limitations

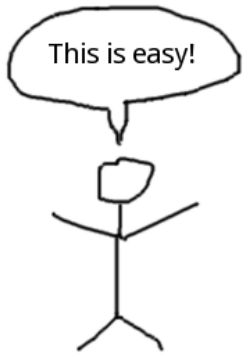
- Curved domains/boundaries using higher order mappings and manifold descriptions
- Arbitrary geometry



Limitations:

- Only regular refinement
- Limited to quads/hexas
- Coarse mesh duplicated on all nodes

In Practice



How to use?

- Replace `Triangulation` by `parallel::distributed::Triangulation`
- Continue to load or create meshes as usual
- Adapt with `GridRefinement::refine_and_coarsen*` and `tr.execute_coarsening_and_refinement()`, etc.
- You can only look at own cells and ghost cells: `cell->is_locally_owned()`, `cell->is_ghost()`, or `cell->is_artificial()`
- Of course: dealing with DoFs and linear algebra changes!

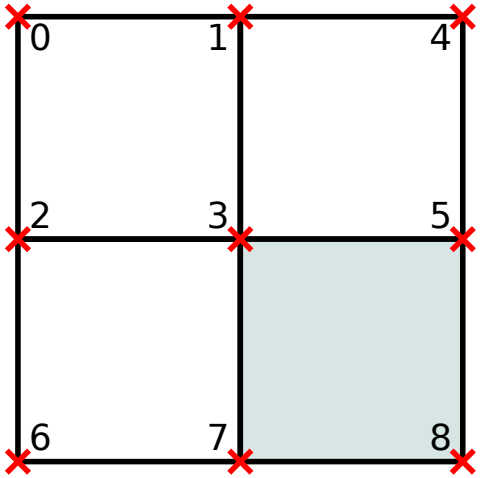
Meshes in deal.II

	serial mesh	dynamic parallel mesh	static parallel mesh
name	Triangulation	parallel::distributed ::Triangulation	(just an idea)
duplicated	everything	coarse mesh	nothing
partitioning	METIS	p4est: fast, scalable	offline, (PAR)METIS?
part. quality	good	okay	good?
hp?	yes	(planned)	yes?
geom. MG?	yes	in progress	?
Aniso. ref.?	yes	no	(offline only)
Periodicity	yes	yes	?
Scalability	100 cores	16k+ cores	?

parallel::shared::Triangulation will address some shortcomings of "serial mesh": do not duplicate linear algebra, same API as parallel::distributed, ...

Distributing the Degrees of Freedom (DoFs)

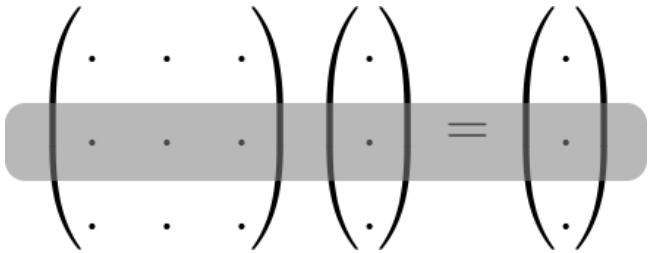
- Create global numbering for all DoFs
- Reason: identify shared ones
- Problem: no knowledge about the whole mesh



Sketch:

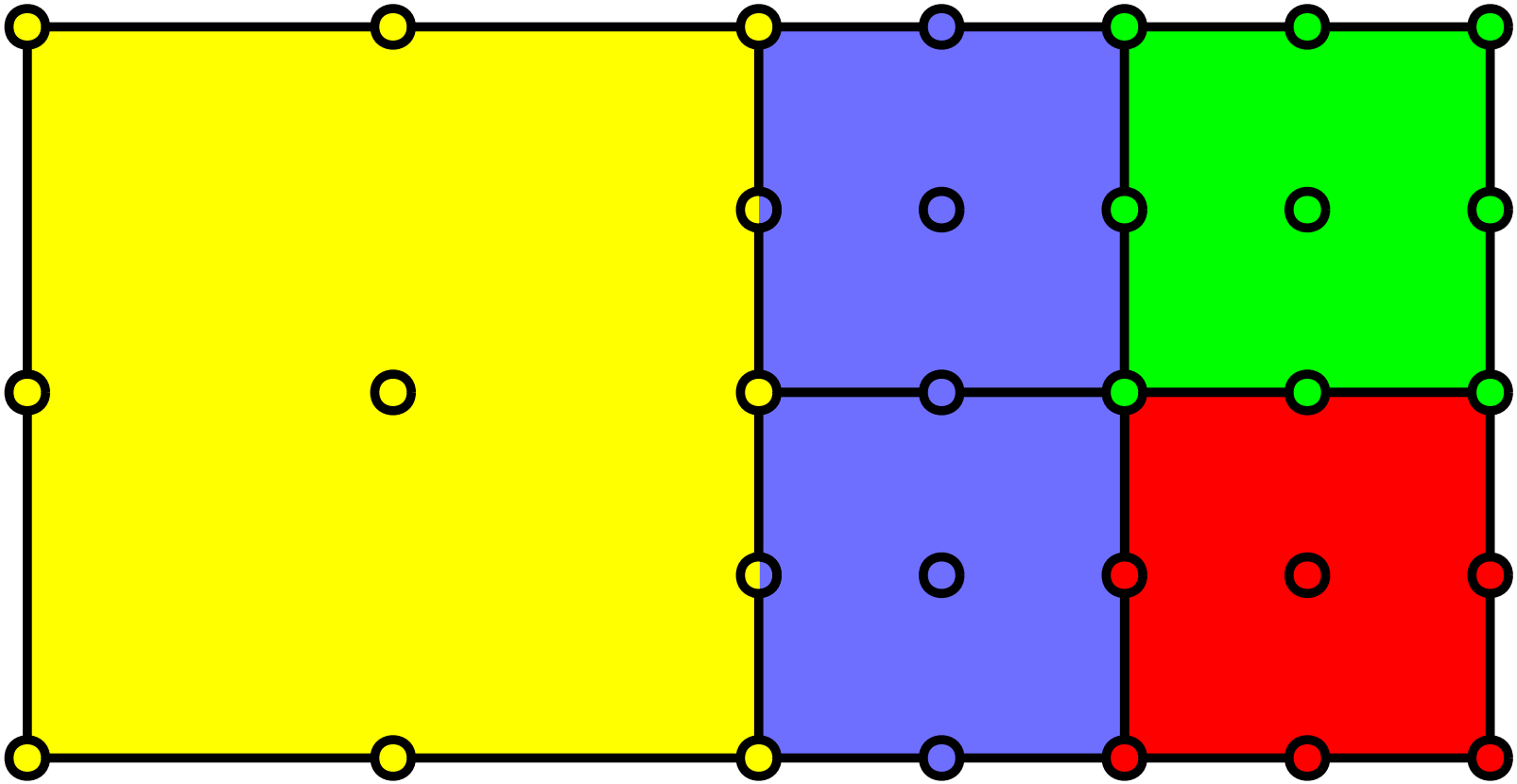
1. Decide on ownership of DoFs on interface (no communication!)
2. Enumerate locally (only own DoFs)
3. Shift indices to make them globally unique (only communicate local quantities)
4. Exchange indices to ghost neighbors

Linear Algebra: Short Version



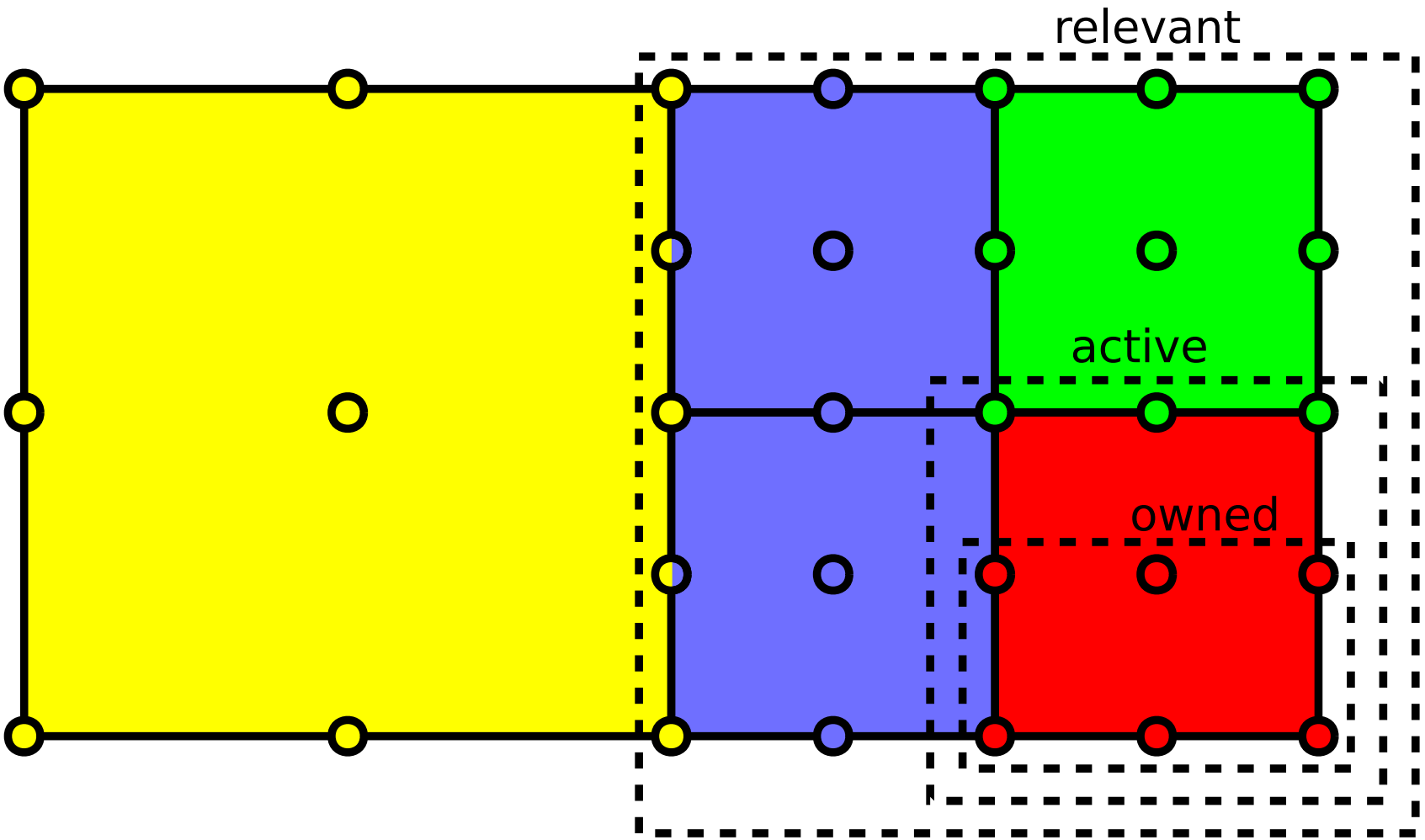
- Use distributed matrices and vectors
- Assemble local parts (some communication on interfaces)
- Iterative solvers (CG, GMRES, . . .) are equivalent, only need:
  - Matrix-vector products
  - scalar products
- Preconditioners:
  - always problem dependent
  - similar to serial: block factorizations, Schur complement approximations
  - not enough: combine preconditioners on each node
  - good: algebraic multigrid
  - in progress: geometric multigrid

Longer Version



- Example: Q2 element and ownership of DoFs
- What might red CPU be interested in?

Longer Version: Interesting DoFs



(perspective of the red CPU)

DoF Sets

- Each CPU has sets:
  - `owned`: we store vector and matrix entries of these rows
  - `active`: we need those for assembling, computing integrals, output, etc.
  - `relevant`: error estimation
- These set are subsets of $\{0, \ldots,$`n_global_dofs`$\}$
- Represented by objects of type `IndexSet`
- How to get? `DoFHandler::locally_owned_dofs()`, `DoFTools::extract_locally_relevant_dofs()`, `DoFHandler::locally_owned_dofs_per_processor()`, ...

Vectors/Matrices

- reading from owned rows only (for both vectors and matrices)
- writing allowed everywhere (more about `compress` later)
- what if you need to read others?
- Never copy a whole vector to each machine!
- instead: ghosted vectors

Ghosted Vectors

- read-only
- create using
  `Vector(IndexSet owned, IndexSet ghost, MPI_COMM)`
  where ghost is `relevant` or `active`
- copy values into it by using operator=(Vector)
- then just read entries you need

Compressing Vectors/Matrices

- Why?
  - After writing into foreign entries communication has to happen
  - All in one go for performance reasons
- How?
  - `object.compress (VectorOperation::add);` if you added to entries
  - `object.compress (VectorOperation::insert);` if you set entries
  - This is a collective call
- When?
  - After the assembly loop (with `::add`)
  - After you do `vec(j) = k;` or `vec(j) += k;` (and i
    add/insert groups)
  - In no other case (all functions inside deal.II compress if necessary)!

Trilinos vs. PETSc

What should I use?

- Similar features and performance
- Pro Trilinos: more development, some more features (automatic differentation, ...), cooperation with deal.II
- Pro PETSc: stable, easier to compile on older clusters
- But: being flexible would be better! – "why not both?"
  - you can! Example: new `step-40`
  - can switch at compile time
  - need #ifdef in a few places (different solver parameters TrilinosML vs BoomerAMG)
  - some limitations, somewhat work in progress

```
#include <deal.II/lac/generic_linear_algebra.h>
#define USE_PETSC_LA // uncomment this to run with Trilinos

namespace LA
{
#ifdef USE_PETSC_LA
  using namespace dealii::LinearAlgebraPETSc;
#else
  using namespace dealii::LinearAlgebraTrilinos;
#endif
}

// ...
    LA::MPI::SparseMatrix system_matrix;
    LA::MPI::Vector solution;

// ...
    LA::SolverCG solver(solver_control, mpi_communicator);
    LA::MPI::PreconditionAMG preconditioner;

    LA::MPI::PreconditionAMG::AdditionalData data;

#ifdef USE_PETSC_LA
    data.symmetric_operator = true;
#else
    //trilinos defaults are good
#endif
    preconditioner.initialize(system_matrix, data);

// ...
```

Postprocessing, . . .

Not covered today:

- Error estimation
- Decide over refinement and coarsening (communication!)
- Handling hanging nodes and other constraints
- Solution transfer (after refinement and repartitioning)
- Parallel I/O
- and probably more

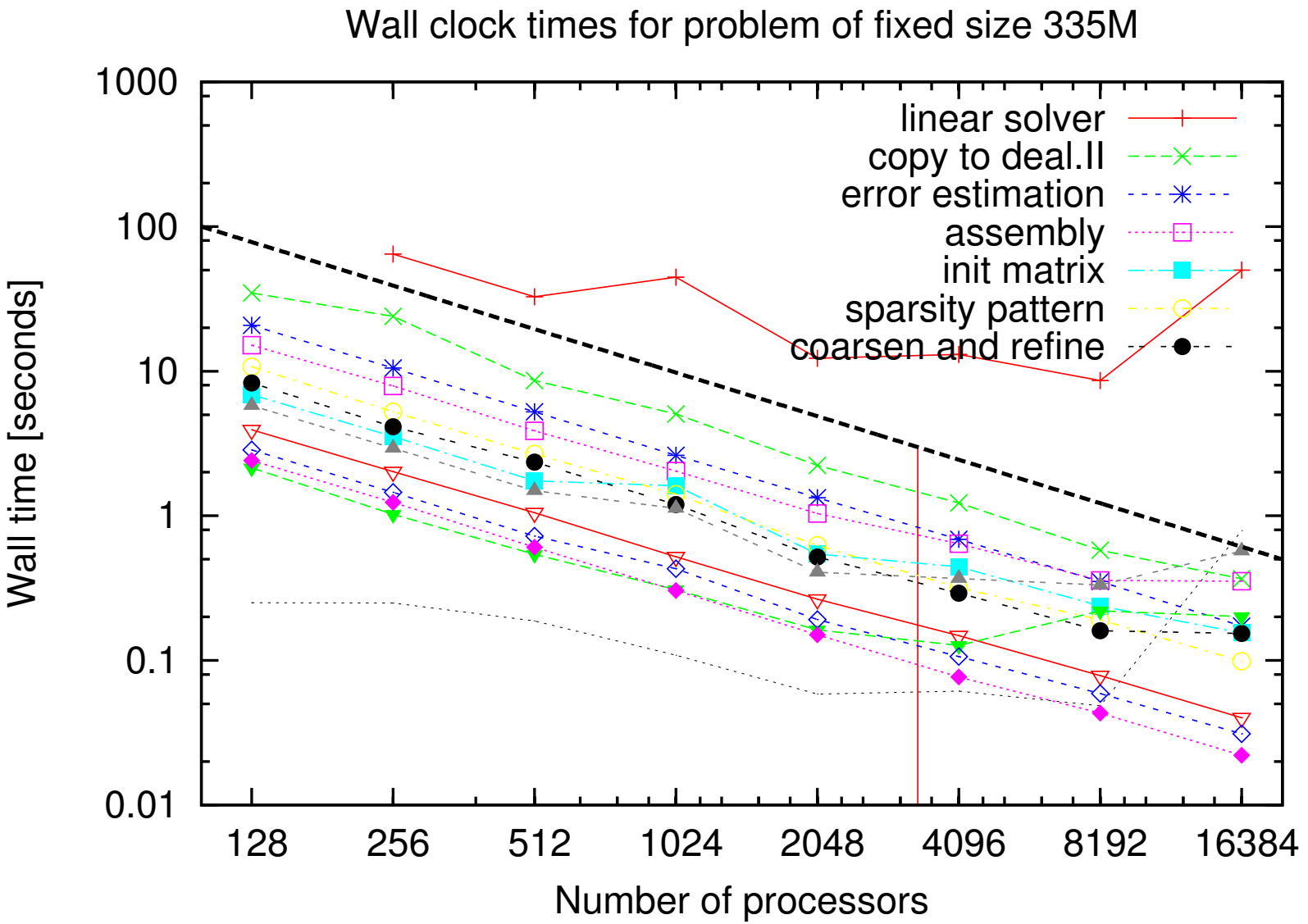
My workflow

1. Write code in serial to test ideas, use UMFPACK
2. Switch to use MPI using namespace LA with direct solver (SparseDirect) – [can we remove this step?]
3. Linear solver: Schur complement based block preconditioner with AMG for each block
4. Profit!

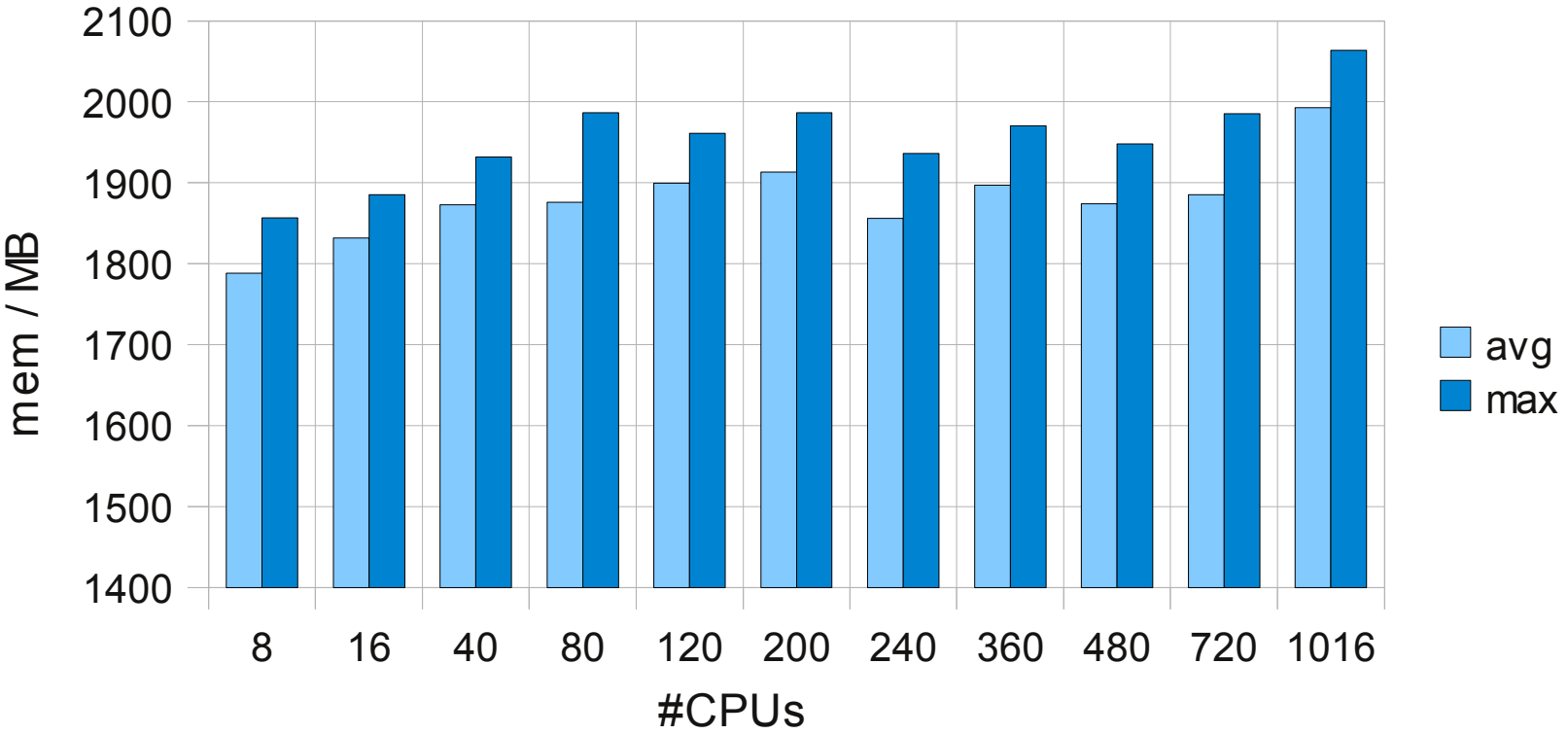
Wishlist

- Unify and simplify linear algebra? Include serial matrices/vectors?
- Need to incorporate even more backends (Tpetra, new PETSc, ...)
- Make something like namespace LA the default? Use inheritance instead?
- Need: different API for writing into vectors because of multithreading
- Our block matrices don't mesh well with PETSc

Strong Scaling: 2d Adaptive Poisson Problem



Test: Memory Consumption

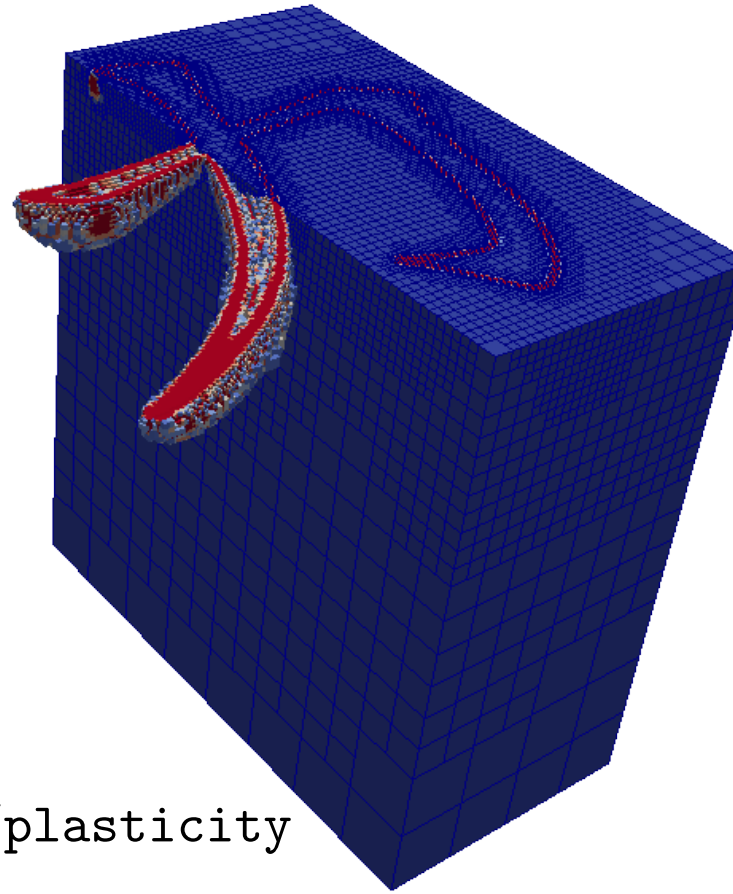


average and maximum memory consumption (VmPeak)
3D, weak scalability from 8 to 1000 processors with about 500.000 DoFs per processor (4 million up to 500 million total)

$\rightsquigarrow$ **Constant memory usage with increasing # CPUs & problem size**

Plasticity

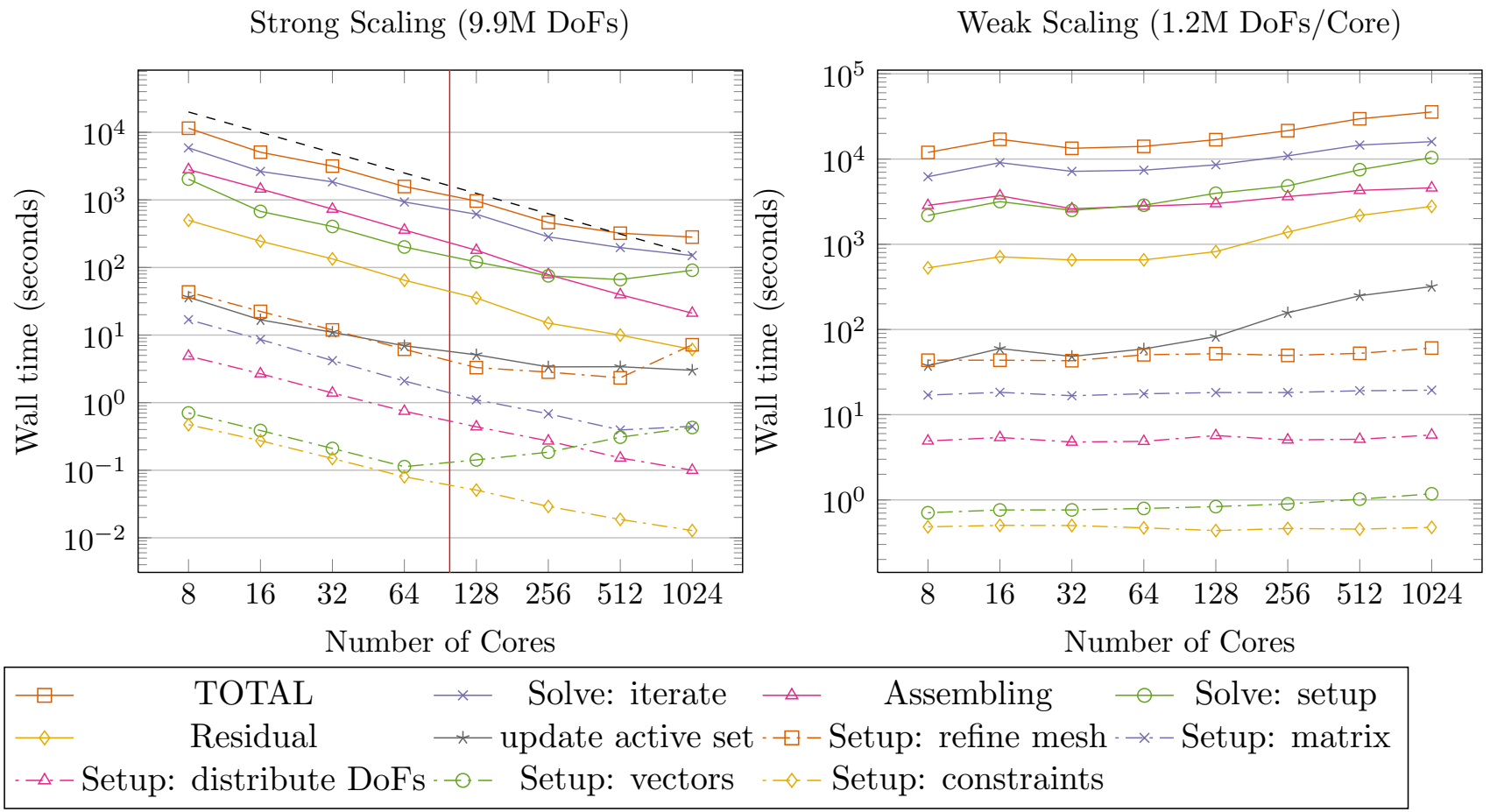
- 3d contact problem
- elasto-plastic material
- isotropic hardening
- semi-smooth Newton
  + active set strategy



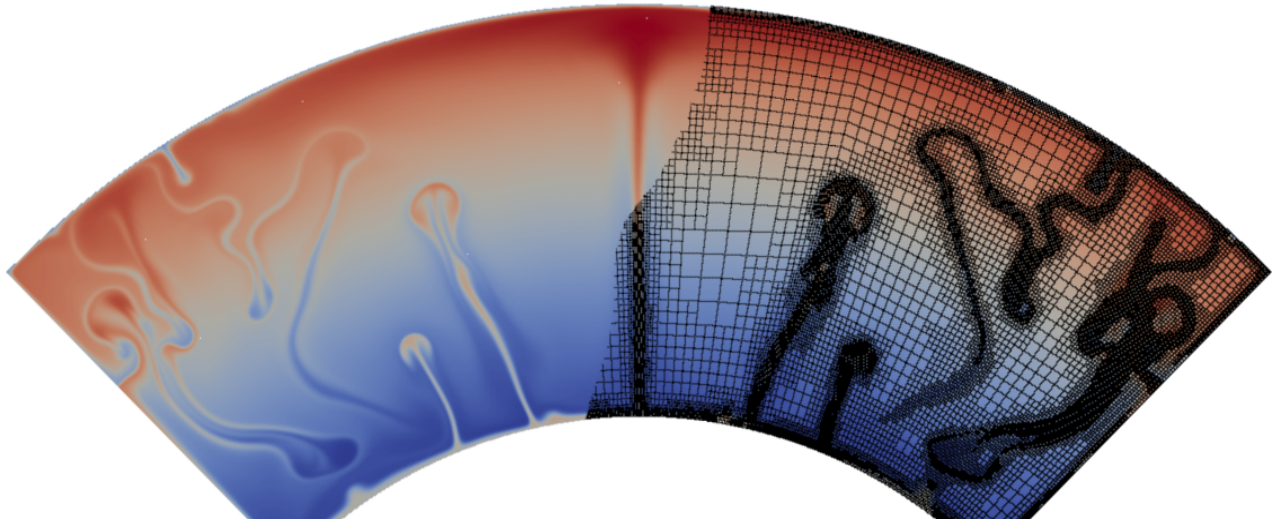
code: step-42 or `https://github.com/tjhei/plasticity`

Frohne, Heister, Bangerth.
Efficient numerical methods for the large-scale, parallel solution of elastoplastic contact problems.
Accepted for publication in IJNME.

Plasticity: Scalability



ASPECT



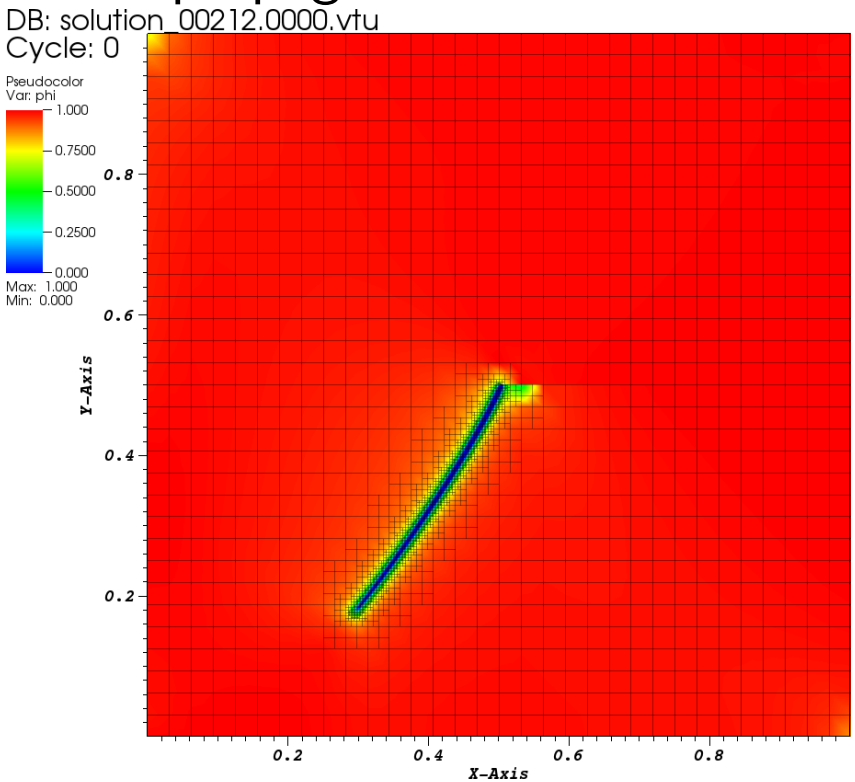
(temperature snapshot, 700,000 degrees of freedom, 2d simulation)

- ASPECT: `http://aspect.dealii.org/`
- Global mantle convection in the Earth's mantle
- 3d computations, adaptive meshes, 100 million+ DoFs
- Need: fast refinement, partitioning
- `https://www.youtube.com/watch?v=iwm68TC5YxM`, `file:aspect.mp4`

Kronbichler, Heister, and Bangerth.
High Accuracy Mantle Convection Simulation through Modern Numerical Methods.
Geophysical Journal International, 2012, 191, 12-29.

Crack propagation

- Crack propagation:

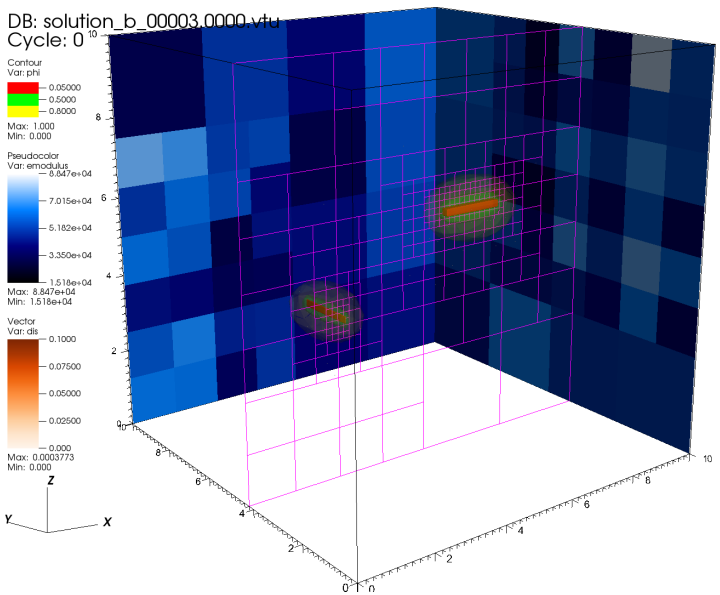


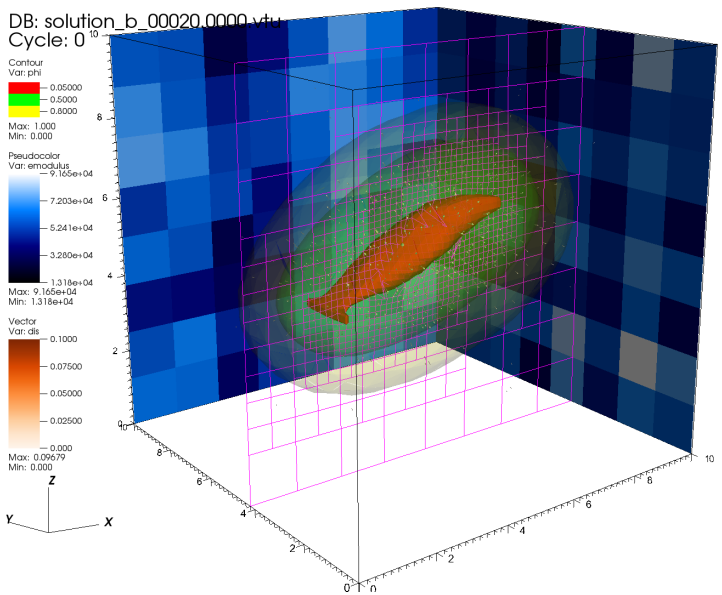
code: https://github.com/tjhei/cracks

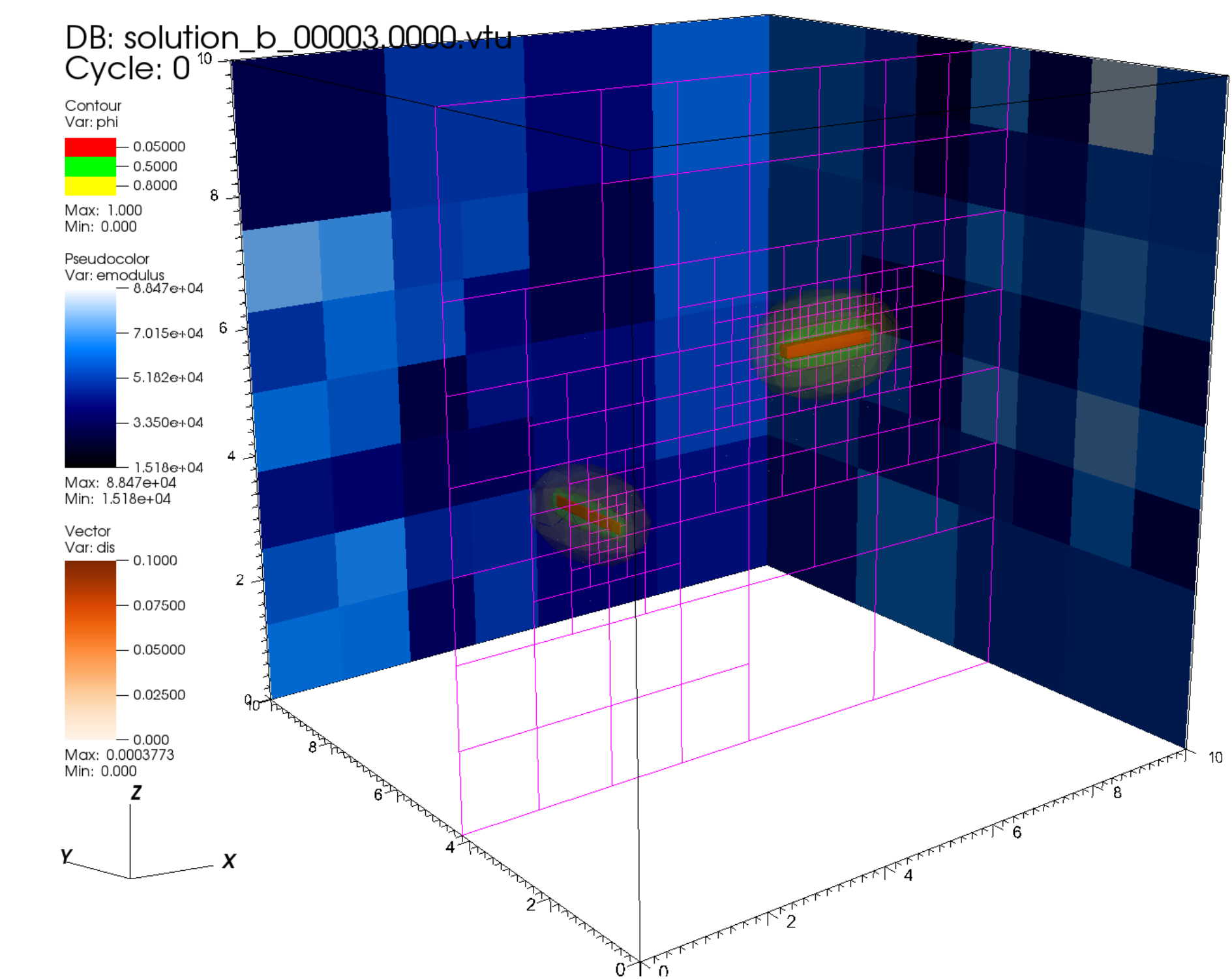
Heister, Wheeler, Wick.
A primal-dual active set method and predictor-corrector mesh adaptivity for computing fracture propagation using a phase-field approach.
CMAME, Volume 290, 15 June 2015

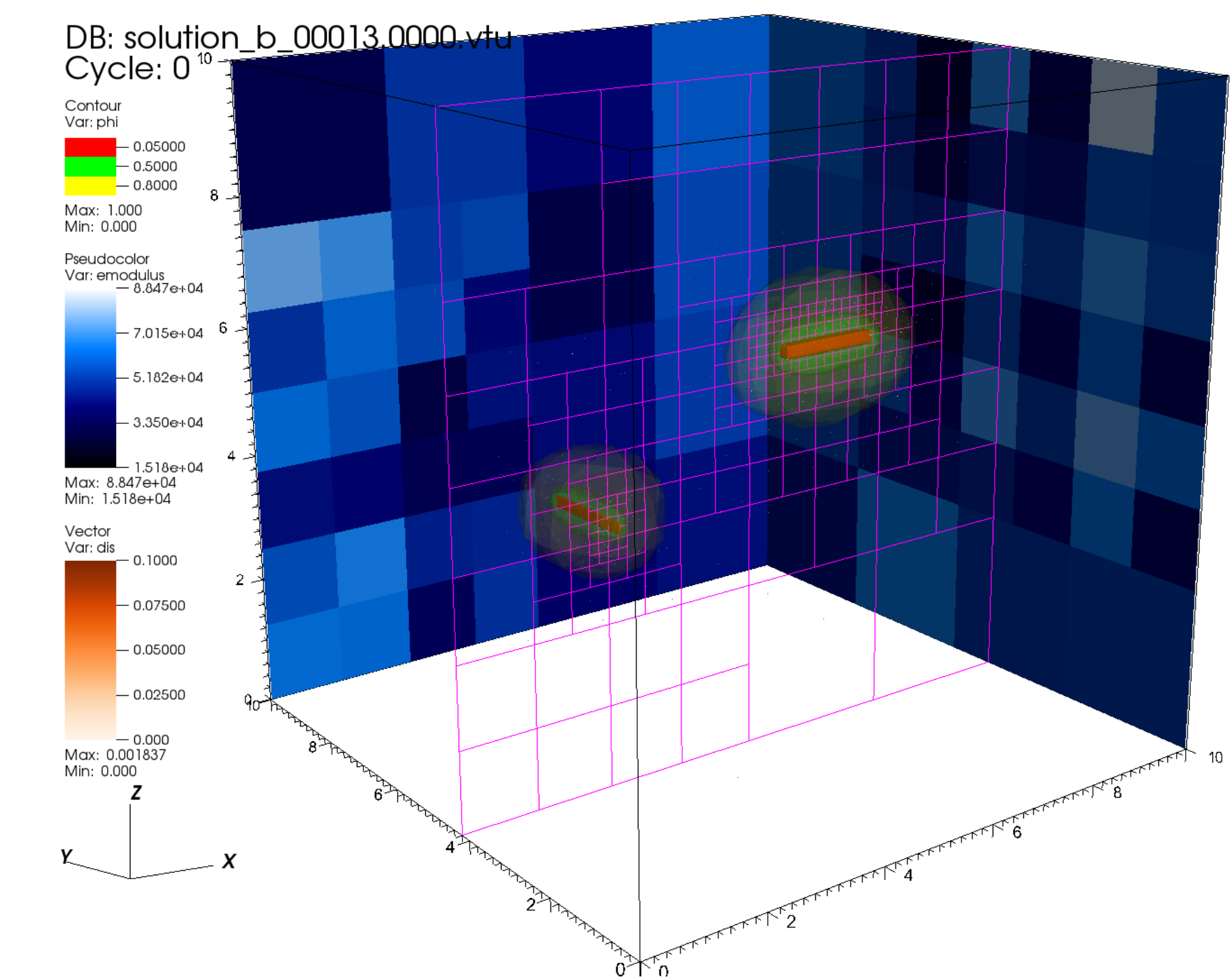
3D Computations

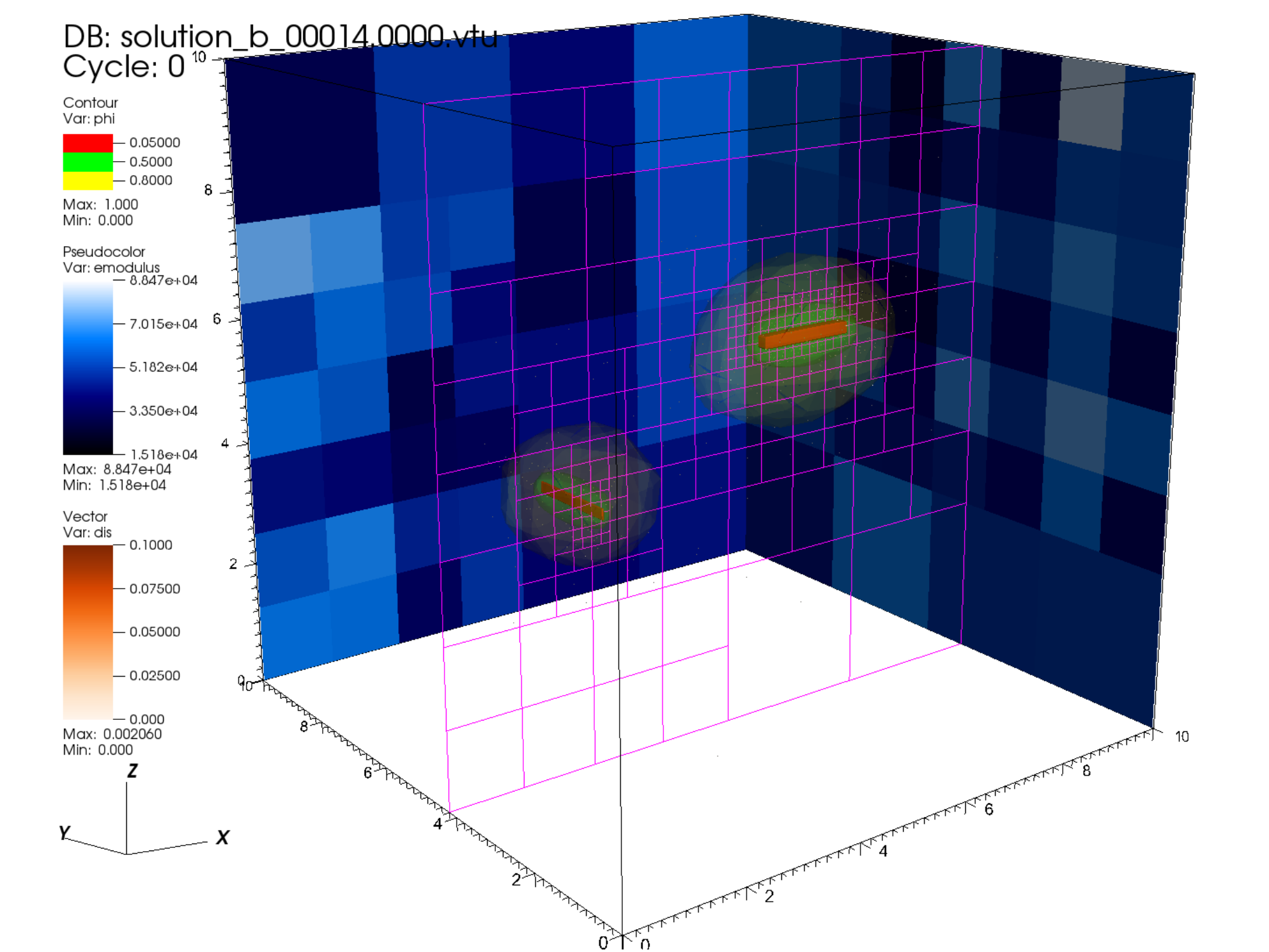
- 3d adaptive test problem
- pressurized crack in heterogeneous medium
- novel adaptive refinement strategy

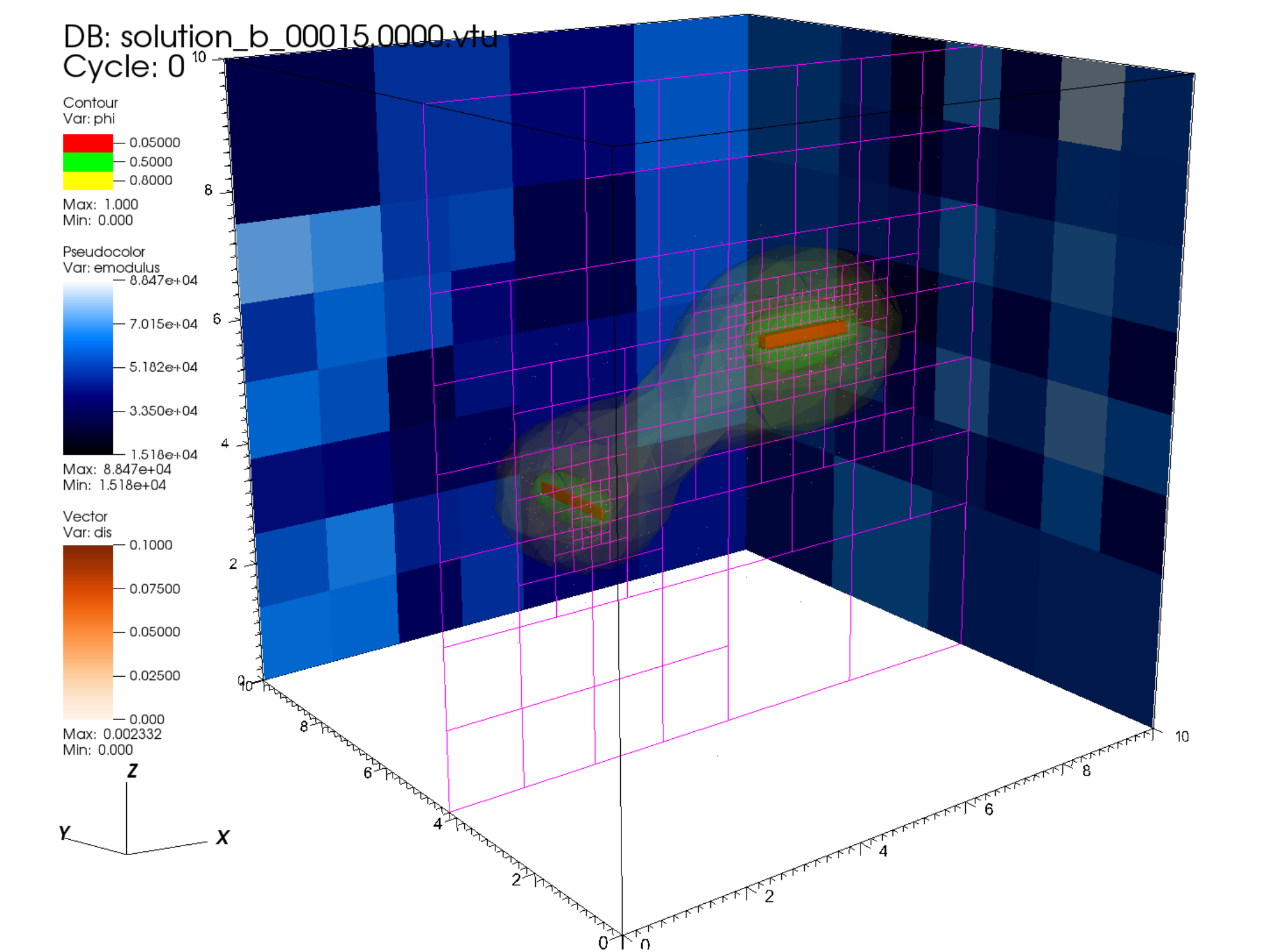


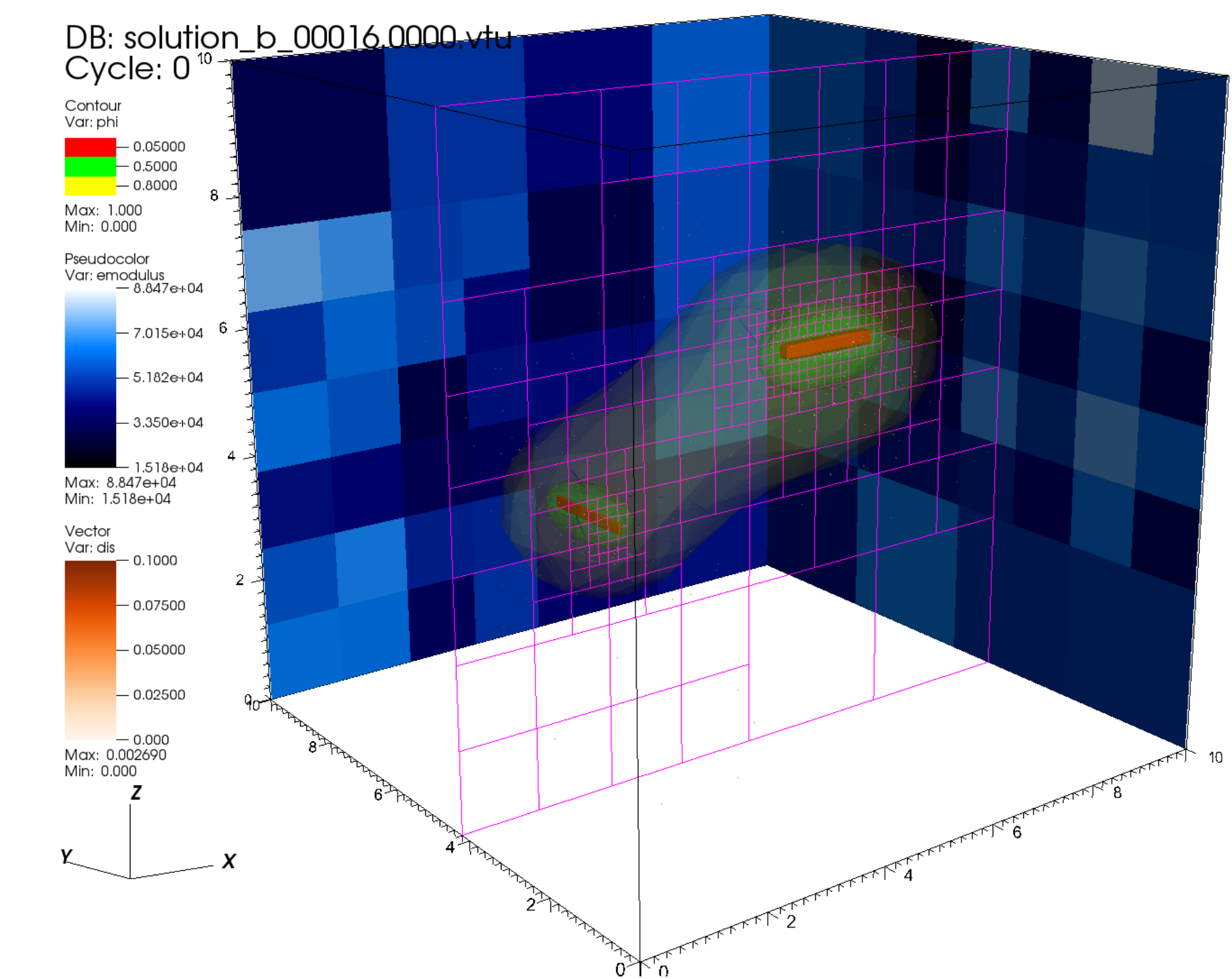


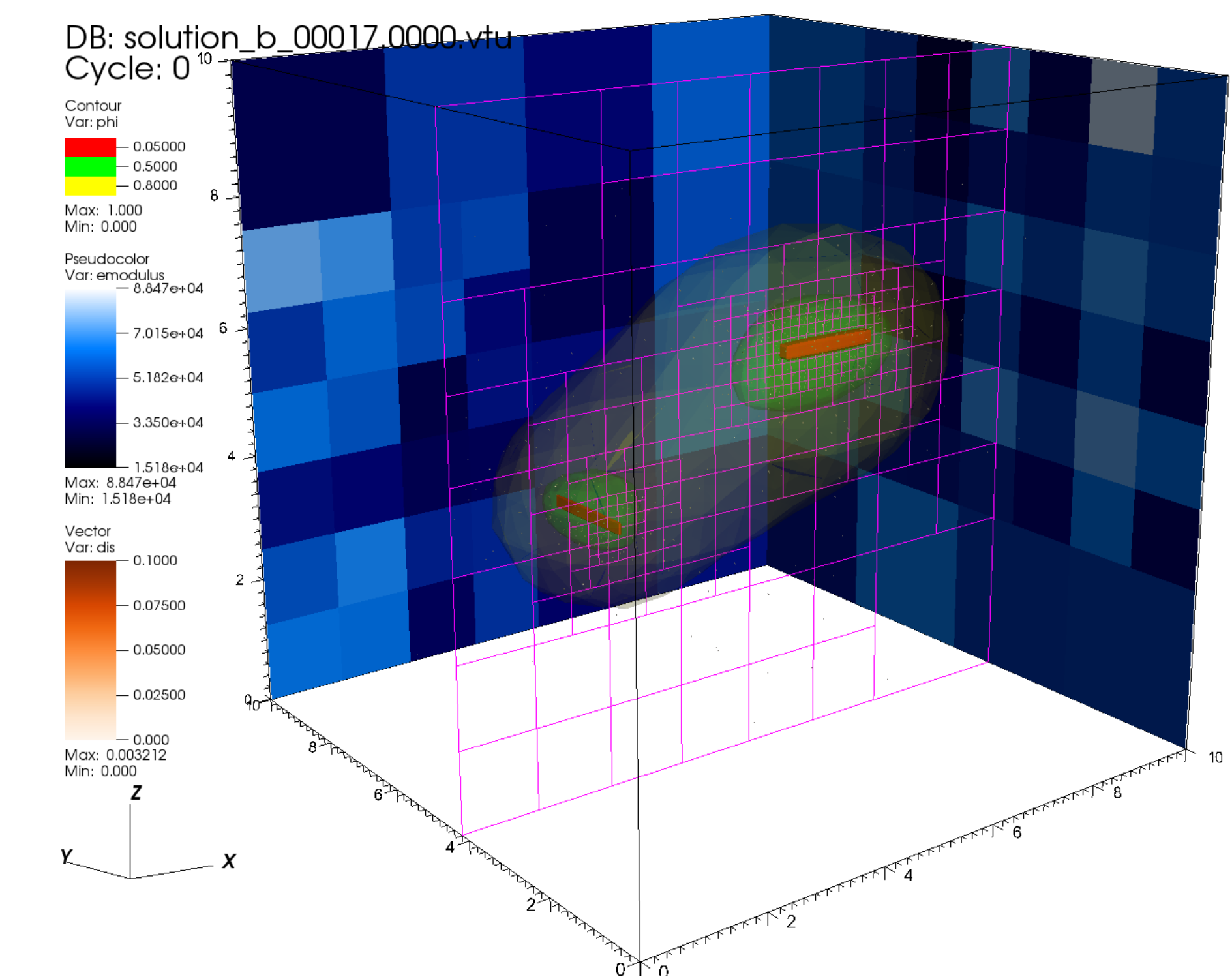

DB: solution_b_00003.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
8.847e+04
7.015e+04
5.182e+04
3.350e+04
1.518e+04
Max: 8.847e+04
Min: 1.518e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.0003773
Min: 0.000
Z
Y
X

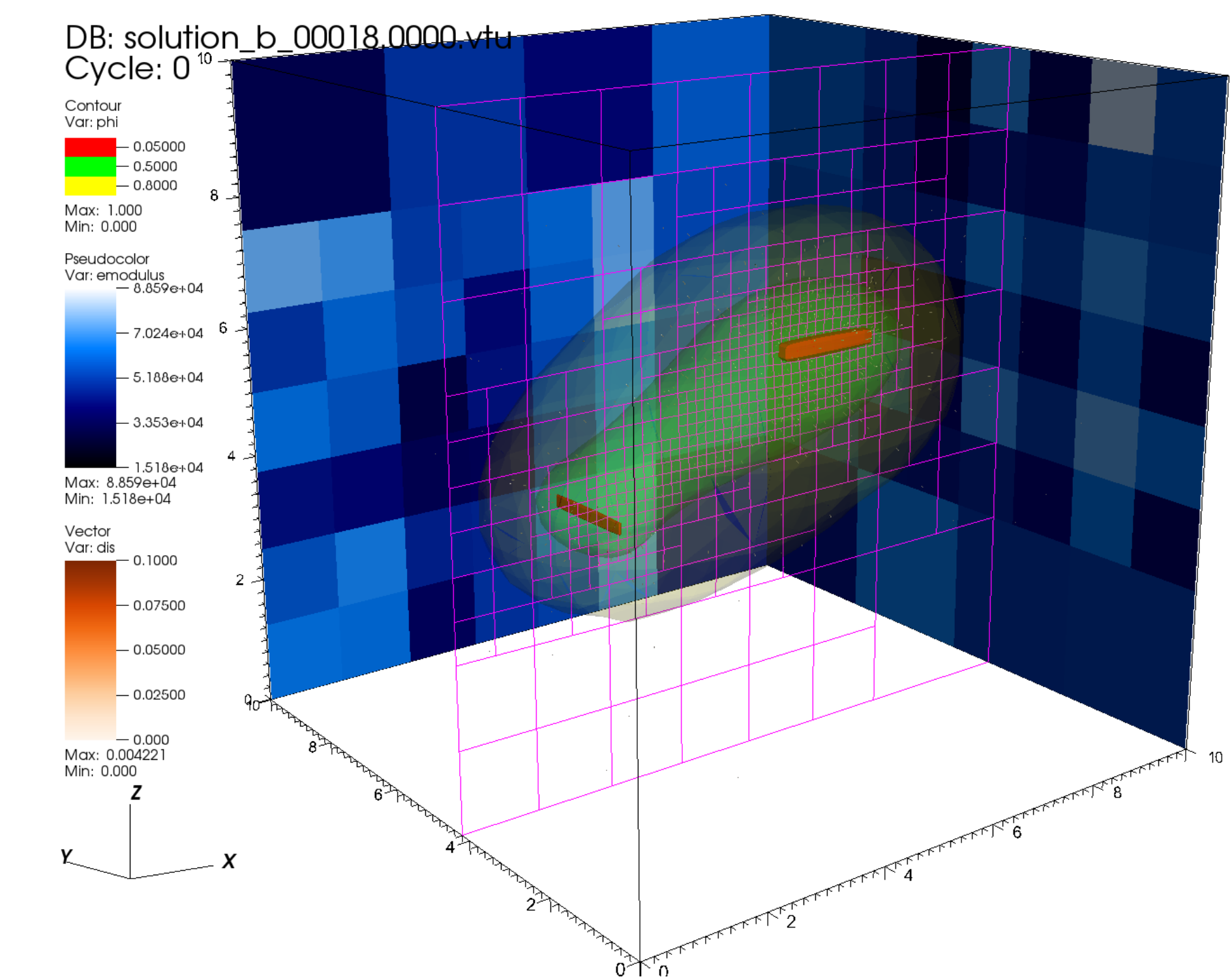

DB: solution_b_00013.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
8.847e+04
7.015e+04
5.182e+04
3.350e+04
1.518e+04
Max: 8.847e+04
Min: 1.518e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.001837
Min: 0.000
Z
Y
X

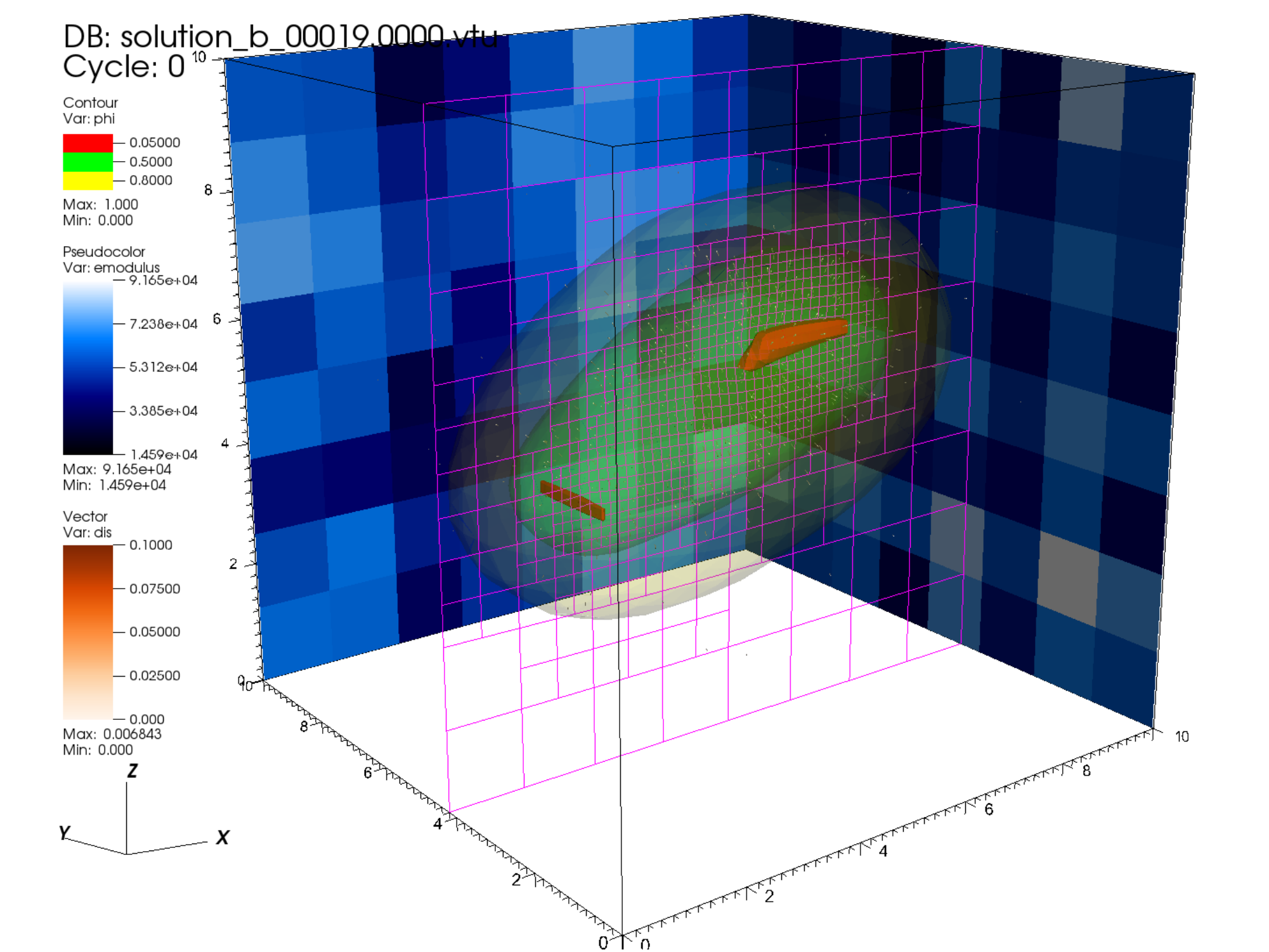

DB: solution_b_00014.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
8.847e+04
7.015e+04
5.182e+04
3.350e+04
1.518e+04
Max: 8.847e+04
Min: 1.518e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.002060
Min: 0.000
Z
Y
X
0
2
4
6
8
10

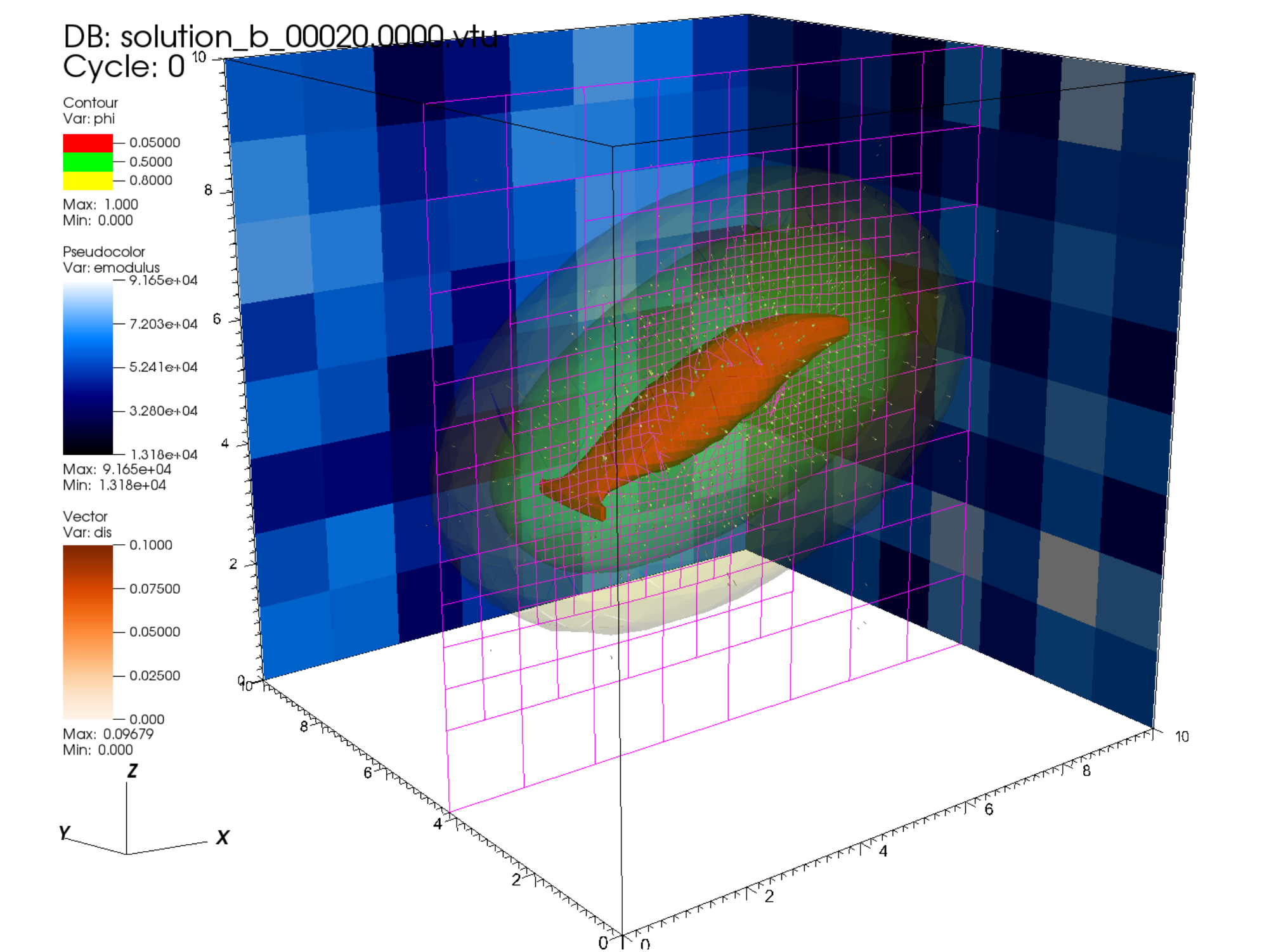

DB: solution_b_00015.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
8.847e+04
7.015e+04
5.182e+04
3.350e+04
1.518e+04
Max: 8.847e+04
Min: 1.518e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.002332
Min: 0.000
Z
Y
X
0
2
4
6
8
10

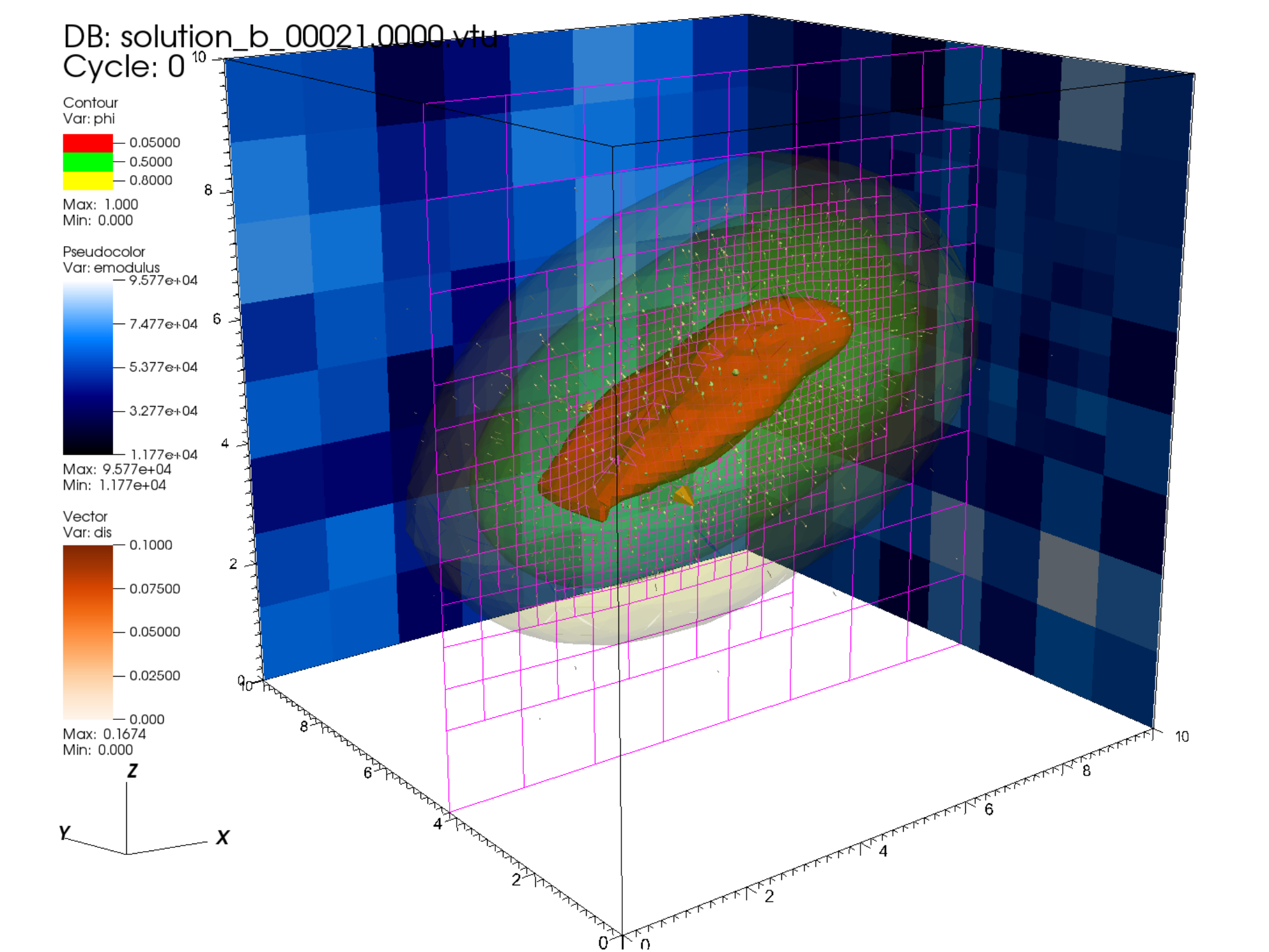

DB: solution_b_00016.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
8.847e+04
7.015e+04
5.182e+04
3.350e+04
1.518e+04
Max: 8.847e+04
Min: 1.518e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.002690
Min: 0.000
Z
Y
X

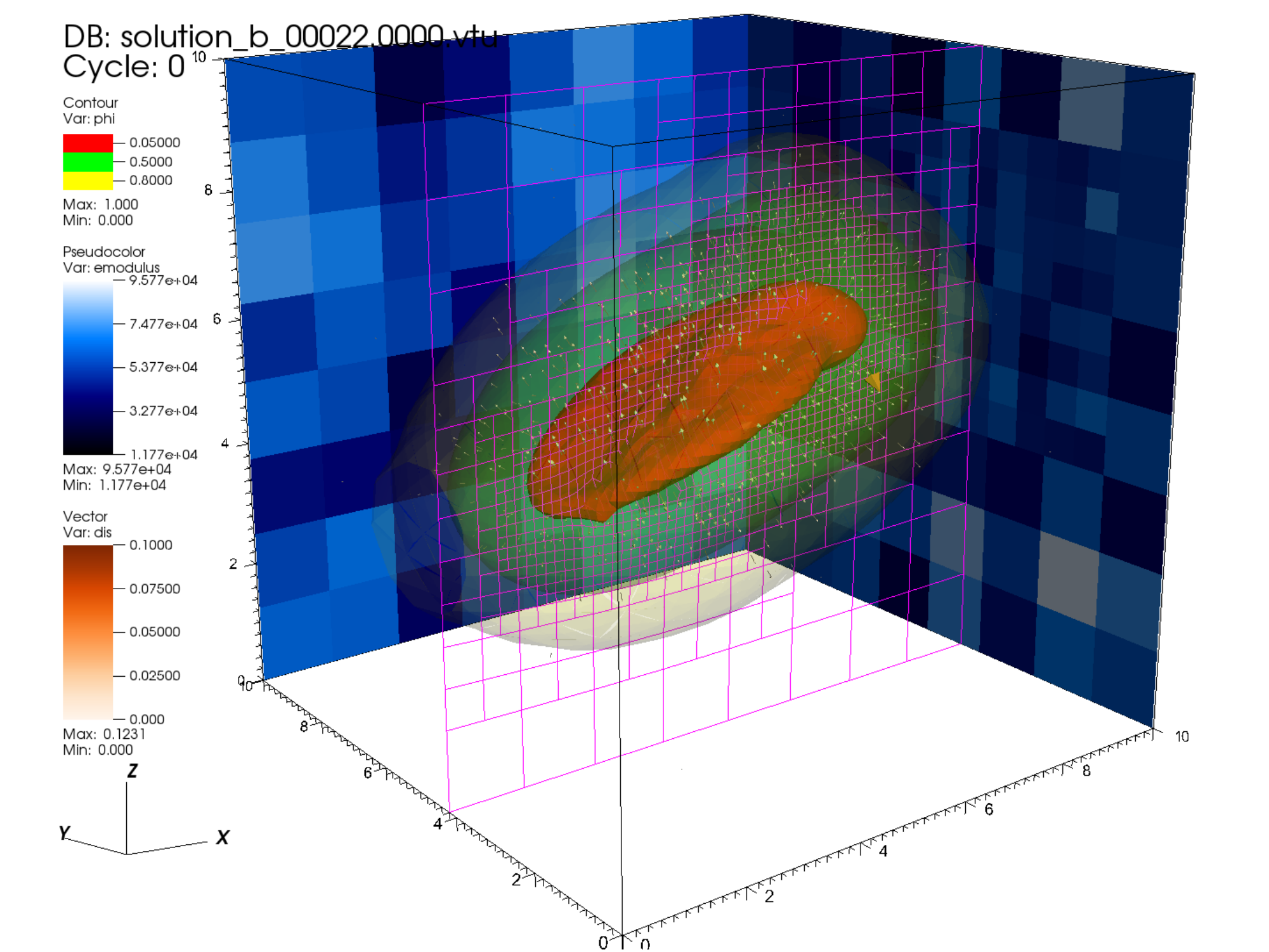

DB: solution_b_00017.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
8.847e+04
7.015e+04
5.182e+04
3.350e+04
1.518e+04
Max: 8.847e+04
Min: 1.518e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.003212
Min: 0.000
Z
Y
X

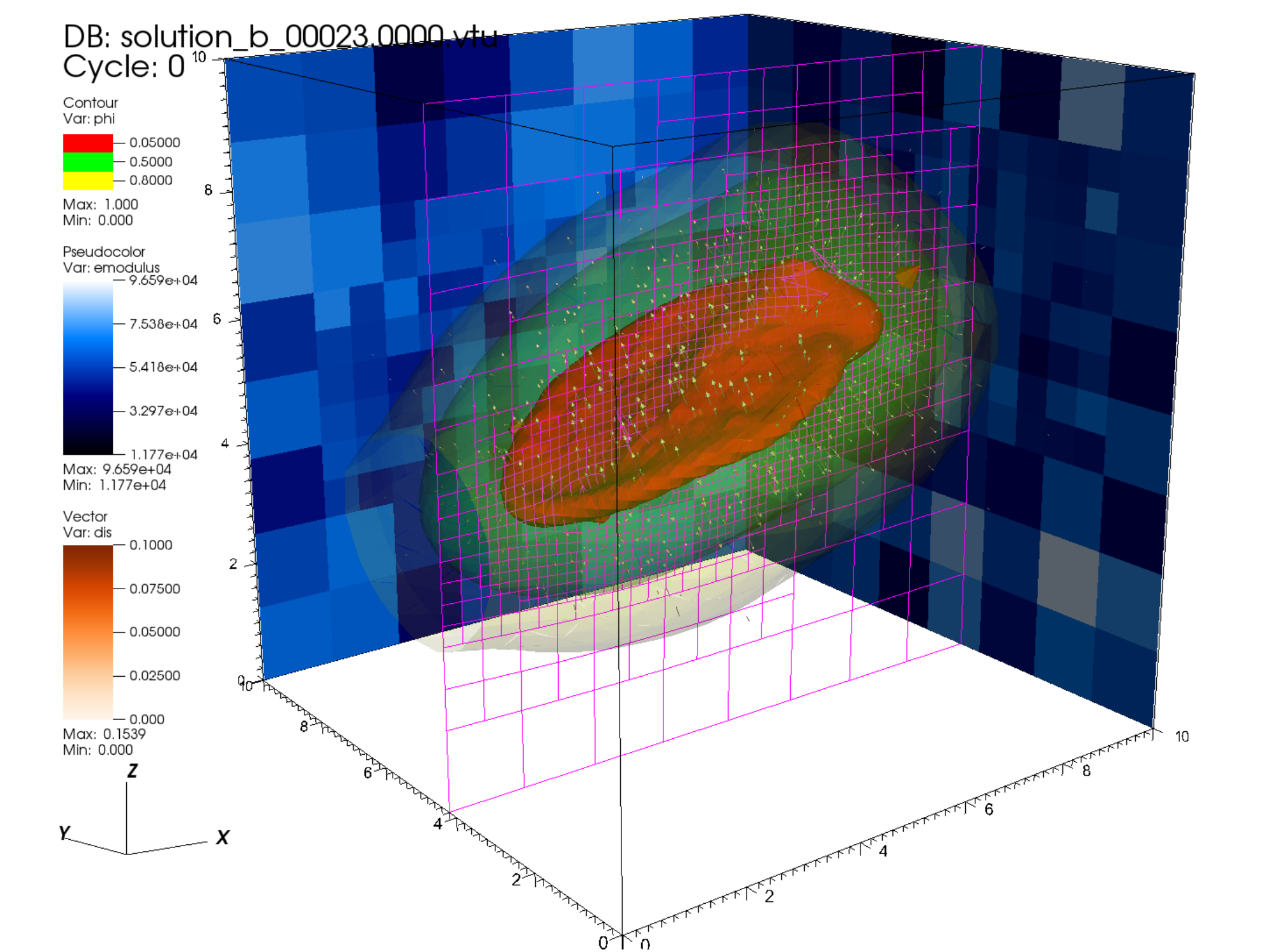

DB: solution_b_00018.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
8.859e+04
7.024e+04
5.188e+04
3.353e+04
1.518e+04
Max: 8.859e+04
Min: 1.518e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.004221
Min: 0.000
Z
Y
X

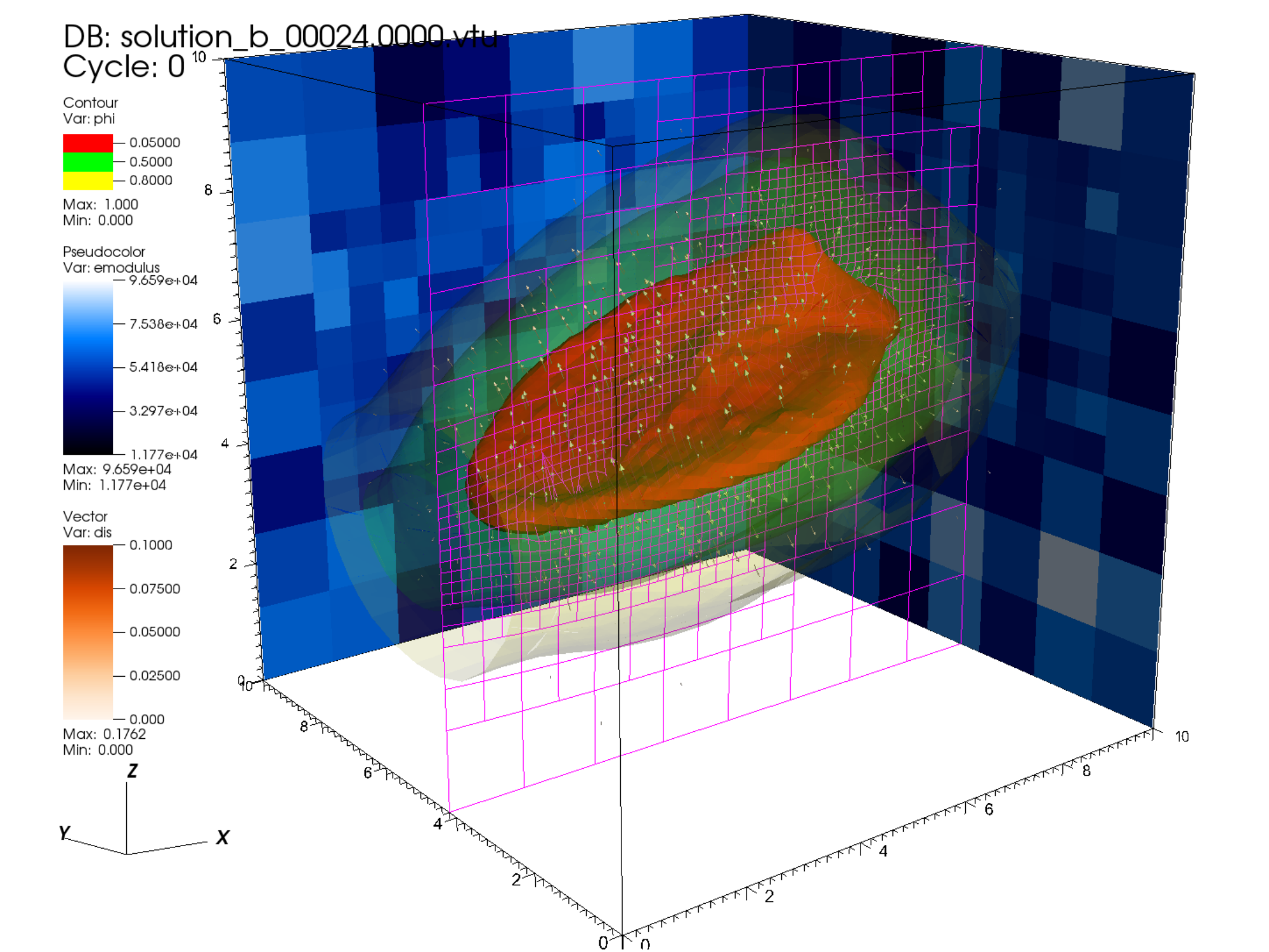

DB: solution_b_00019.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
9.165e+04
7.238e+04
5.312e+04
3.385e+04
1.459e+04
Max: 9.165e+04
Min: 1.459e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.006843
Min: 0.000
Z
Y
X
0
2
4
6
8
10

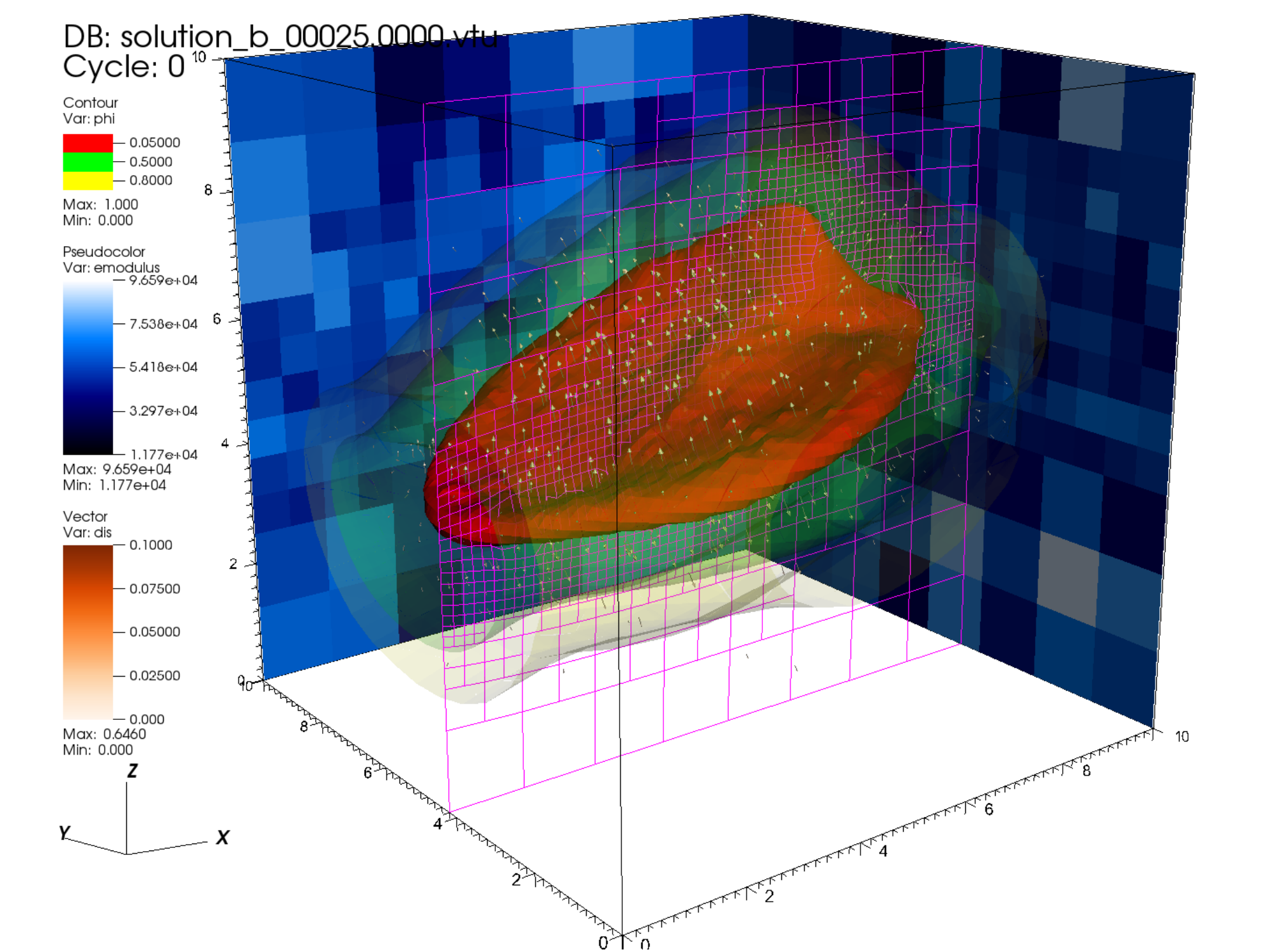

DB: solution_b_00020.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
9.165e+04
7.203e+04
5.241e+04
3.280e+04
1.318e+04
Max: 9.165e+04
Min: 1.318e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.09679
Min: 0.000
Z
Y
X


DB: solution_b_00021.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
9.577e+04
7.477e+04
5.377e+04
3.277e+04
1.177e+04
Max: 9.577e+04
Min: 1.177e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.1674
Min: 0.000
Z
Y
X
0
2
4
6
8
10


DB: solution_b_00022.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
9.577e+04
7.477e+04
5.377e+04
3.277e+04
1.177e+04
Max: 9.577e+04
Min: 1.177e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.1231
Min: 0.000
Z
Y
X
0
2
4
6
8
10

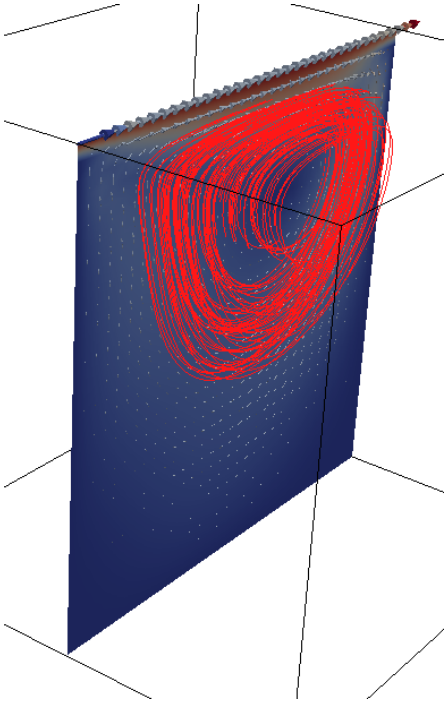

DB: solution_b_00023.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
9.659e+04
7.538e+04
5.418e+04
3.297e+04
1.177e+04
Max: 9.659e+04
Min: 1.177e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.1539
Min: 0.000
Z
Y
X
0
2
4
6
8
10

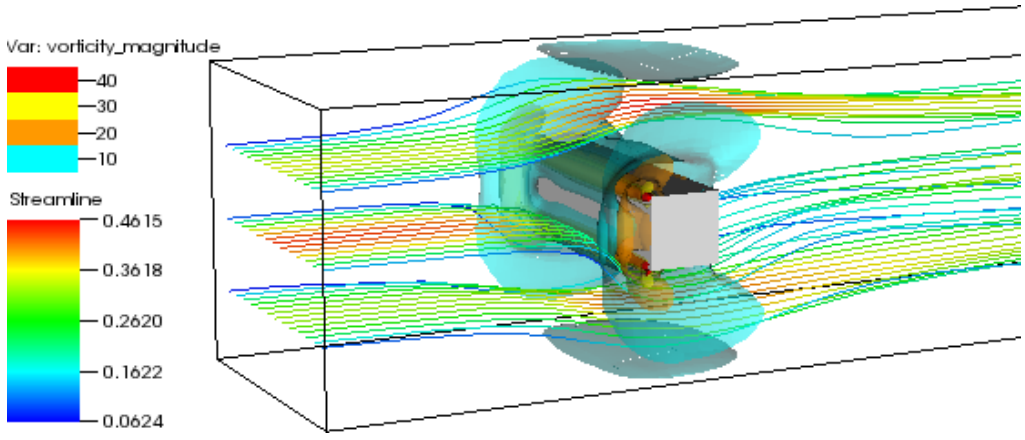

DB: solution_b_00024.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
9.659e+04
7.538e+04
5.418e+04
3.297e+04
1.177e+04
Max: 9.659e+04
Min: 1.177e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.1762
Min: 0.000
Z
Y
X
0
2
4
6
8
10


DB: solution_b_00025.0000.vtu
Cycle: 0
Contour
Var: phi
0.05000
0.5000
0.8000
Max: 1.000
Min: 0.000
Pseudocolor
Var: emodulus
9.659e+04
7.538e+04
5.418e+04
3.297e+04
1.177e+04
Max: 9.659e+04
Min: 1.177e+04
Vector
Var: dis
0.1000
0.07500
0.05000
0.02500
0.000
Max: 0.6460
Min: 0.000
Z
Y
X
0
2
4
6
8
10

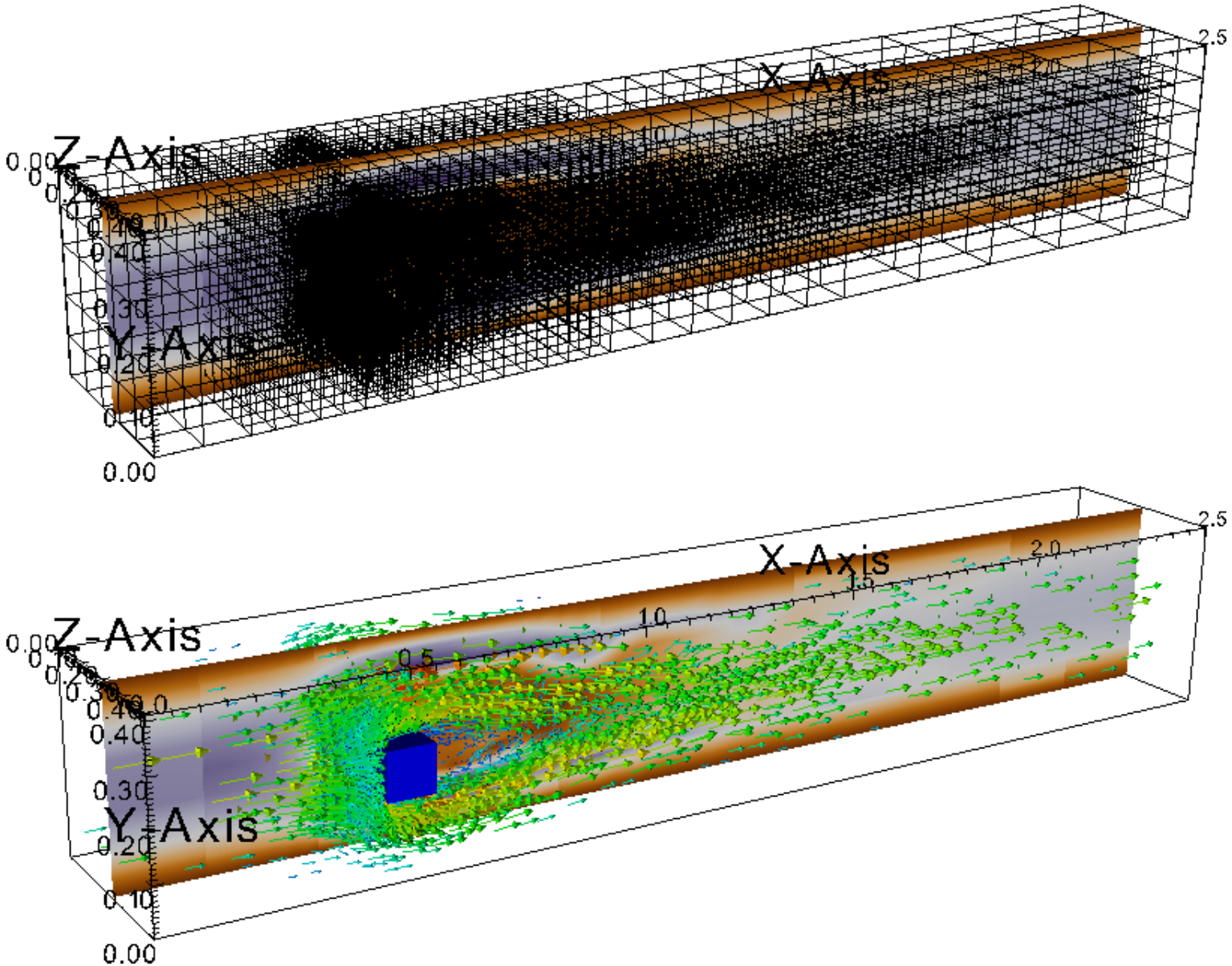
Incompressible Flow (WIP)

- massively parallel, adaptive Incompressible Navier-Stokes solver
- 100+ million DoFs
- efficient linear solvers for coupled system
- even for stationary problems
- Testbed for different discretizations, e.g. velocity-vorticity
- Code: open sourced soon?



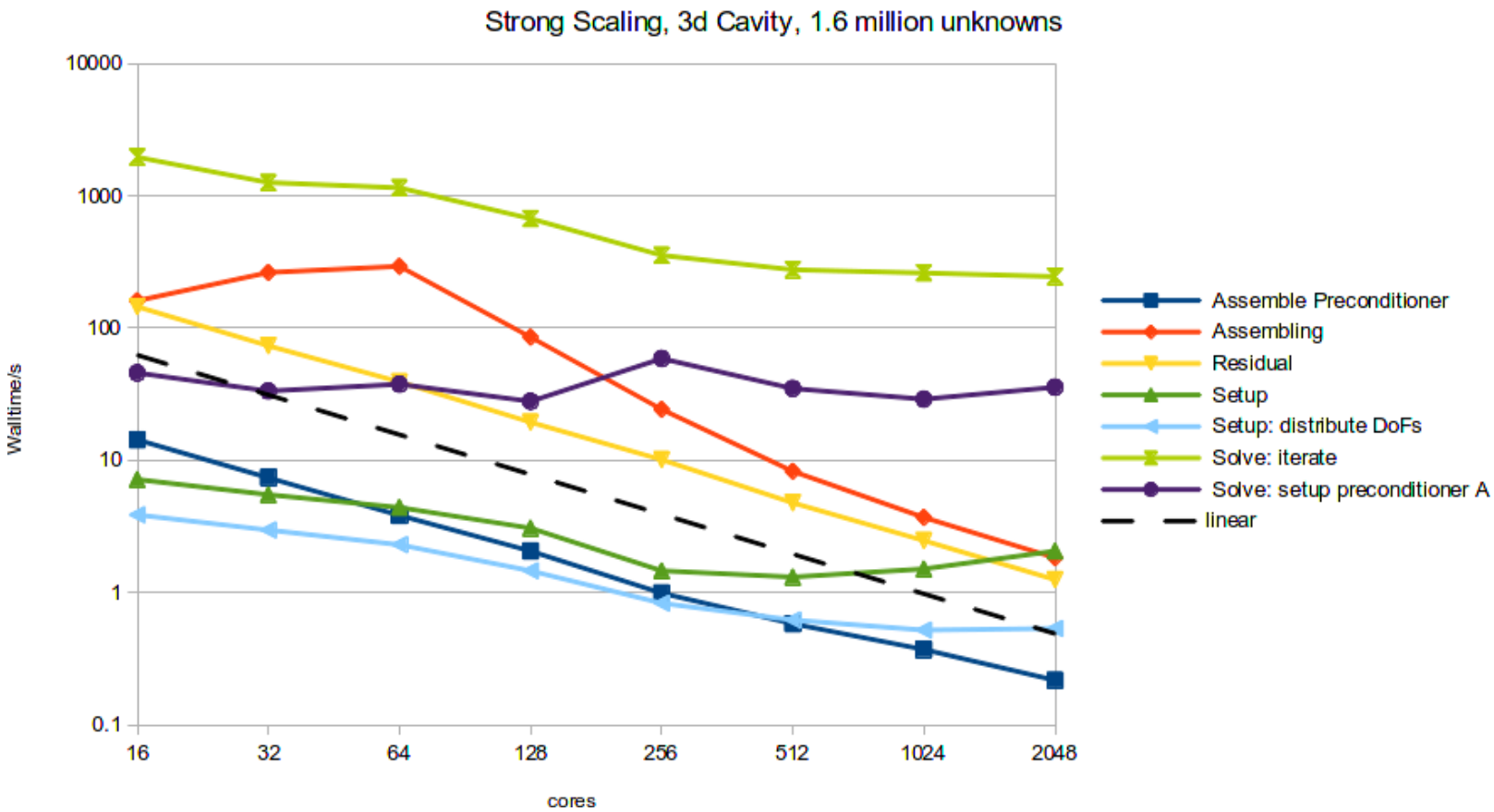


Flow around Cylinder



Scaling

- strong scaling even for tiny test problems:



Geometric Multigrid: Goals

linear solver that

- is generic (different PDEs)
- is flexible (CG, DG, arbitrary order, ...)
- supports adaptive mesh refinement
- is efficient
- is scalable
- works on future architectures

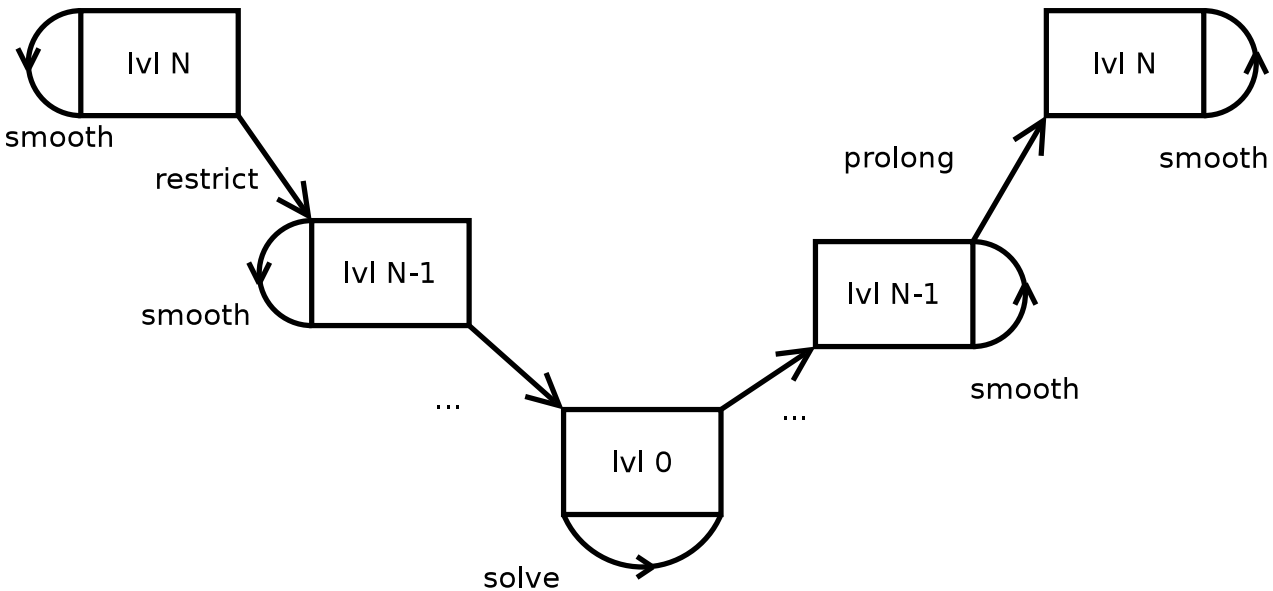
Note: I am ignoring GPUs and I think that is okay.

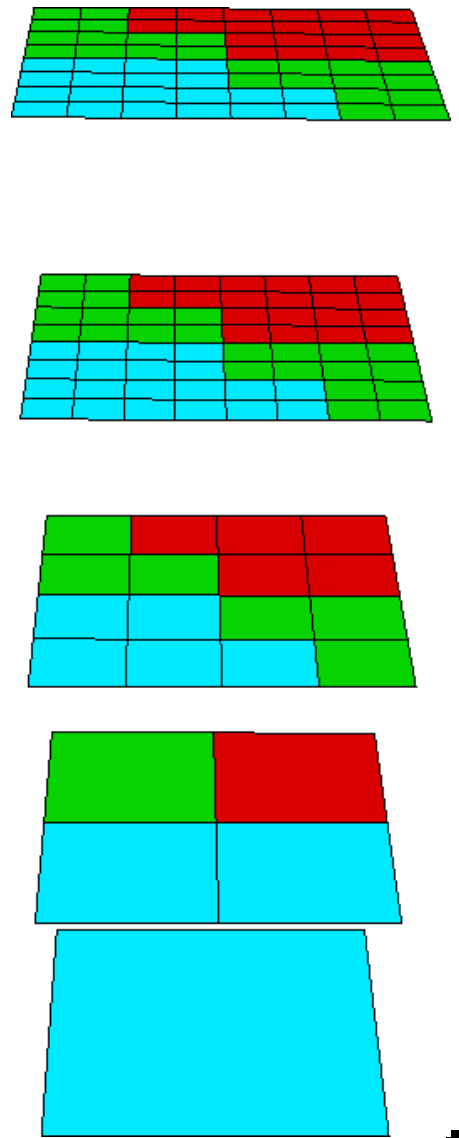
Future Architectures

- likely:
  - many more cores per node
  - more flops per memory bandwith
  - less memory per core
- consequences:
  - higher order elements
  - avoid building sparse matrices
  - hybrid parallelization (MPI+multithreading+vectorization)

Multigrid Intro

- Only method known that is $O(N)$
- Based on hierarchy of levels
- Operations:
  smooth, restrict, coarse solve, prolong





Why not algebraic multigrid?

- always based on sparse matrices
- construct coarser levels by analyzing matrix structure/entries
  $\rightsquigarrow$ communication overhead!
- does not take advantage of geometry structure
- mostly good for poisson type operators
- can not exploit PDE specifics
- but: easy to implement and use (Trilinos ML or Hypre)

Past: geometric Multigrid

- based on local smoothing
- flexible (CG, DG, . . .)
- sparse matrices for levels/restriction/prolongation
- serial implementation only

B. Janssen, G. Kanschat. Adaptive Multilevel Methods with Local Smoothing for H^1- and H^{curl}-Conforming High Order Finite Element Methods. *SIAM J. Sci. Comput.*, vol. 33/4, pp. 2095-2114, 2011.

Past: massively parallel FE

- fully distributed, adaptively refined meshes
- MPI only
- based on sparse matrices, PETSc/Trilinos, AMG
- scalable: 10k+ cores, 4+ billion unknowns

W. Bangerth, C. Burstedde, T. Heister, M. Kronbichler. Algorithms and Data Structures for Massively Parallel Generic Finite Element Codes. *ACM Trans. Math. Softw.*, Volume 38(2), 2011.

Past: matrix free computations

- take advantage of tensor-structure of FE spaces
- using the geometric multigrid framework
- multithreading (Intel TBB)
- (in part explicit) vectorization based on processing n cells at the same time

M. Kronbichler, K. Kormann. A generic interface for parallel cell-based finite element operator application *Computers and Fluids*, vol. 63, pp. 135-147, 2012

Past, Present, and Future

- Past, incompatible:
  - serial GMG
  - distributed meshes
  - matrix free computations
- Now:
  - Parallel geometric multigrid
  - Distributed meshes
  - MPI, based on sparse matrices
- In the future:
  - matrix free
  - hybrid parallel
  - combine with AMG on coarser level

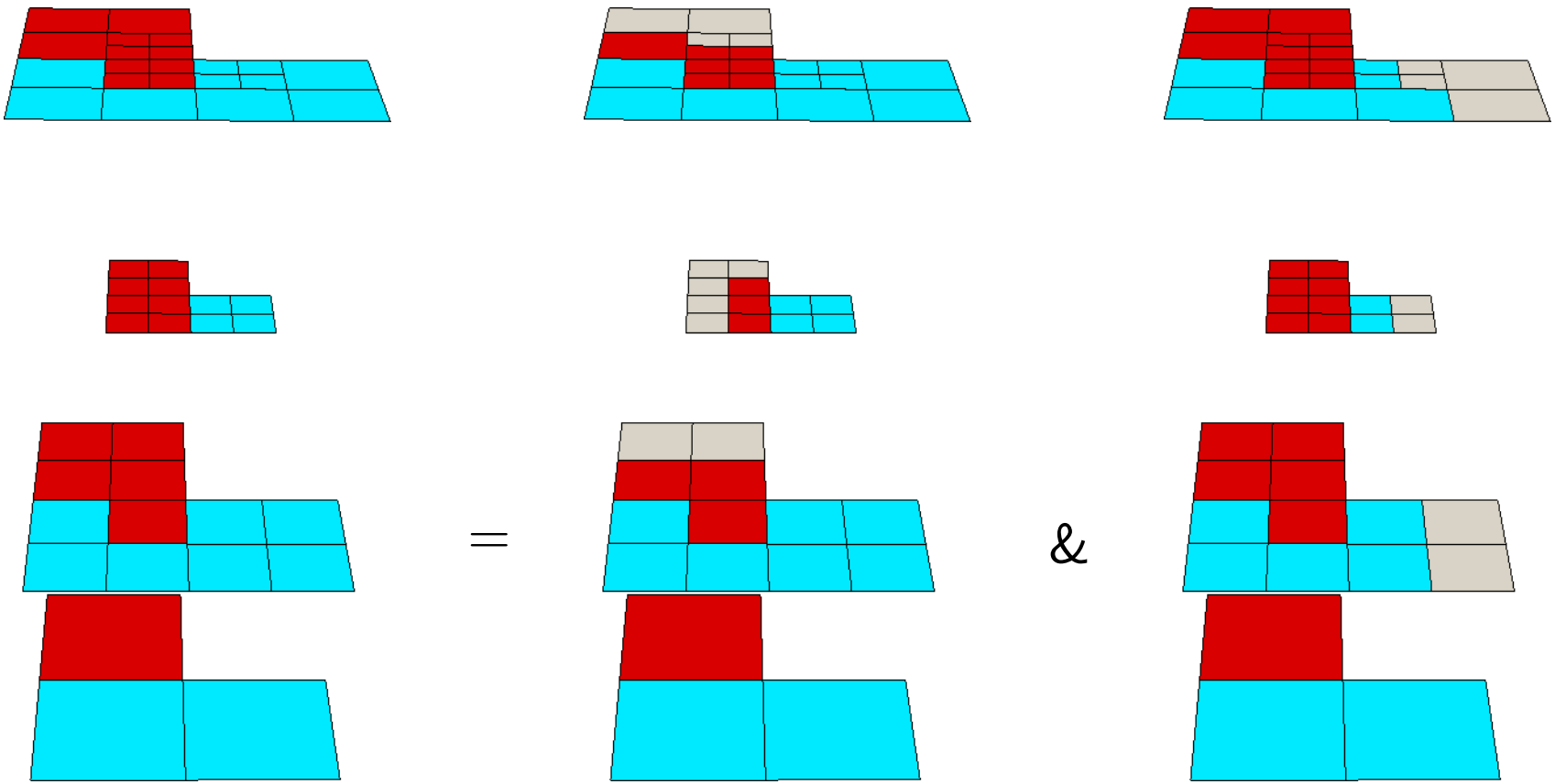
Idea

- equivalent to serial MG
- create distributed transfer and level matrices
- Need: ownership of cells/dofs on each level

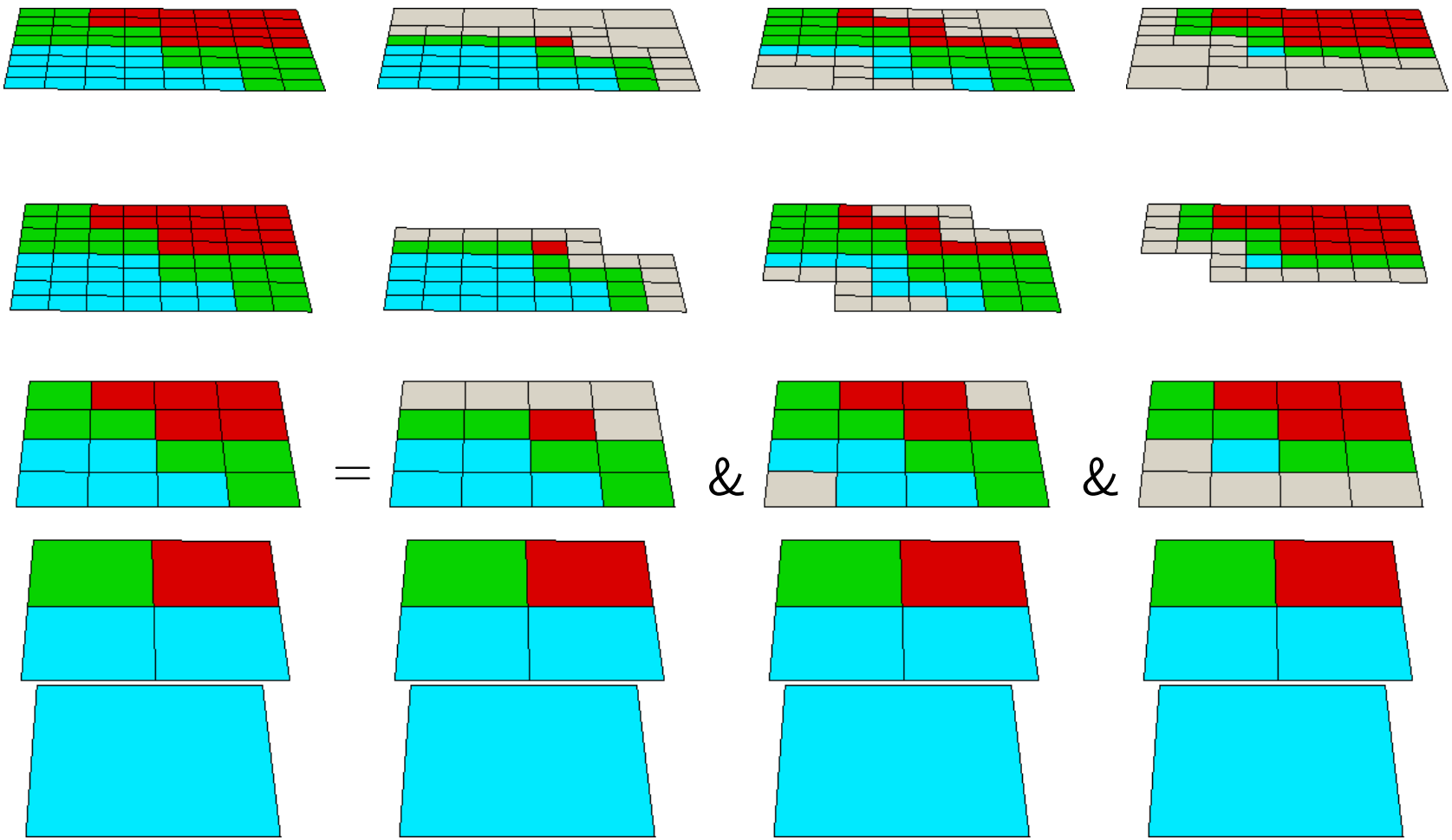
Level cell ownership:

- simplest idea: owner of first child
- need to construct ghost layer

Distribute Level Cells



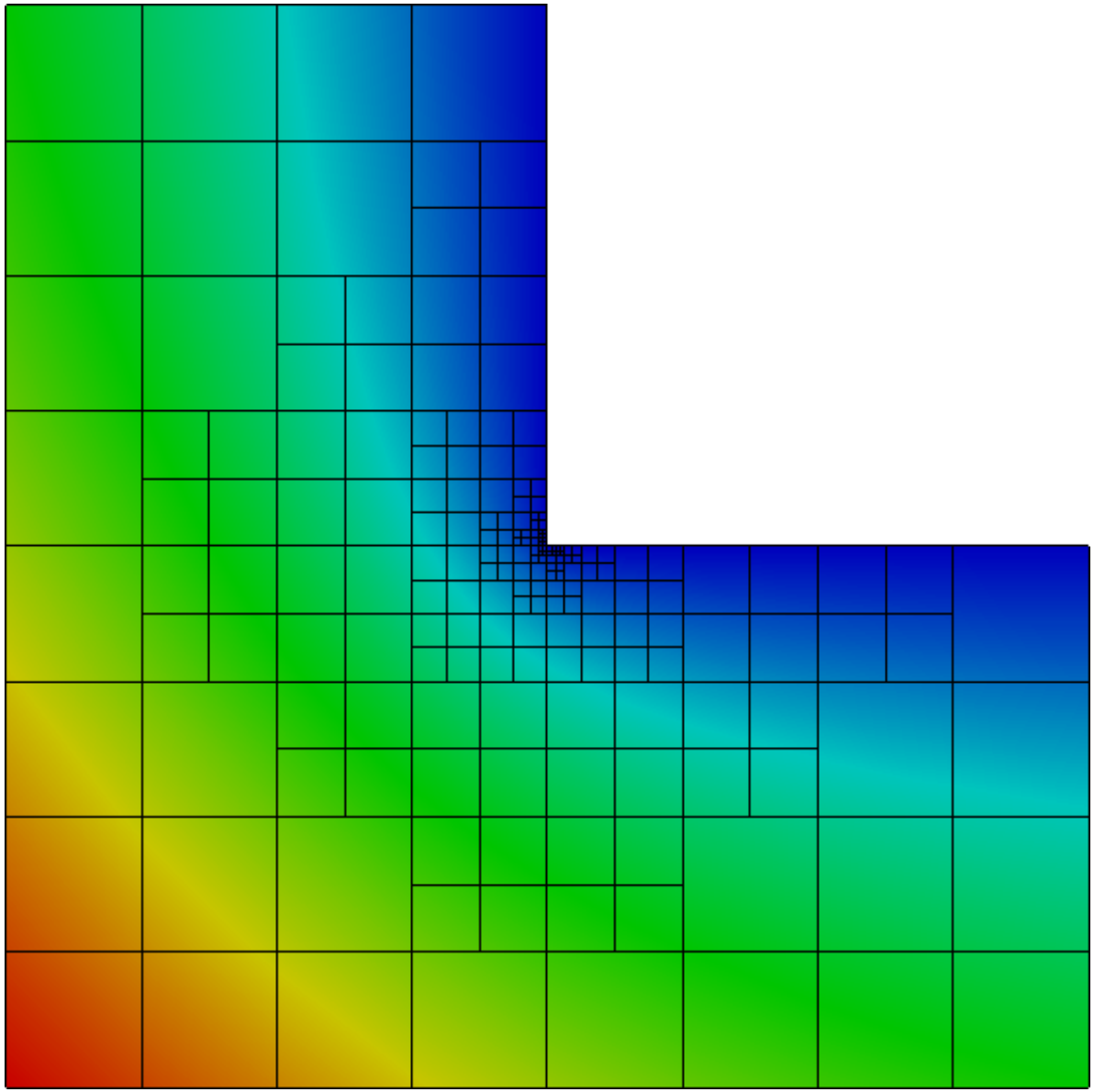
Distribute Level Cells



Distribute Level DoFs

- distribute DoFs locally on each CPU and level
- communicate with ghost neighbors
- difficult: consistent constraints (level boundary DoFs, hanging nodes, . . .)

- L shape, 2d, laplace problem,DGQ2



				iterations			
ref	cells	lvls	dofs	1 CPU	2 CPUs	4 CPUs	8 CPU
1	12	2	108	8	8	8	8
5	45	5	405	9	9	9	9
10	531	10	4779	9	9	9	9
11	897	11	8073	9	9	9	9
12	1521	12	13689	9	9	9	9
13	2553	13	22977	9	9	9	9
14	4413	14	39717	9	9	9	9
15	7533	15	67797	9	9	9	9
16	13005	16	117045	9	9	9	9
17	22749	17	204741	9	9	9	9
18	39063	18	351567	9	9	9	9

(rel. res: 1e-8, CG, 1 V-cycle, jacobi smoother)

Balanced?

Locally refined mesh. Number of cells per level:

lvl	proc0	proc1	proc2	proc3	proc4	proc5	proc6
0	1	1	0	1	0	2	0
1	3	2	3	2	2	6	2
2	11	9	9	11	6	25	9
3	41	38	35	46	23	85	36
4	162	145	132	174	87	145	131
5	647	485	484	652	304	474	466
6	152	232	224	160	311	225	232
7	232	304	324	212	456	284	332
8	40	68	76	32	96	40	76

- Good enough?

Performance?

- Compare with Trilinos ML:

	GMG	AMG
Setup	2s	2s
Assemble	5s	5s
Setup MG	4s	1.5s
Assemble MG	6s	-
Solve	17s	11s
total Solver	27s	12.5s

Note:

- Setup MG does some stupid things
- We go all the way down to 3 cells
- Naive smoother: Jacobi
- Better on large scale?

(Triangulation 113304 cells, 20 levels, 1019736 dofs, 4 cores)

My TODO list

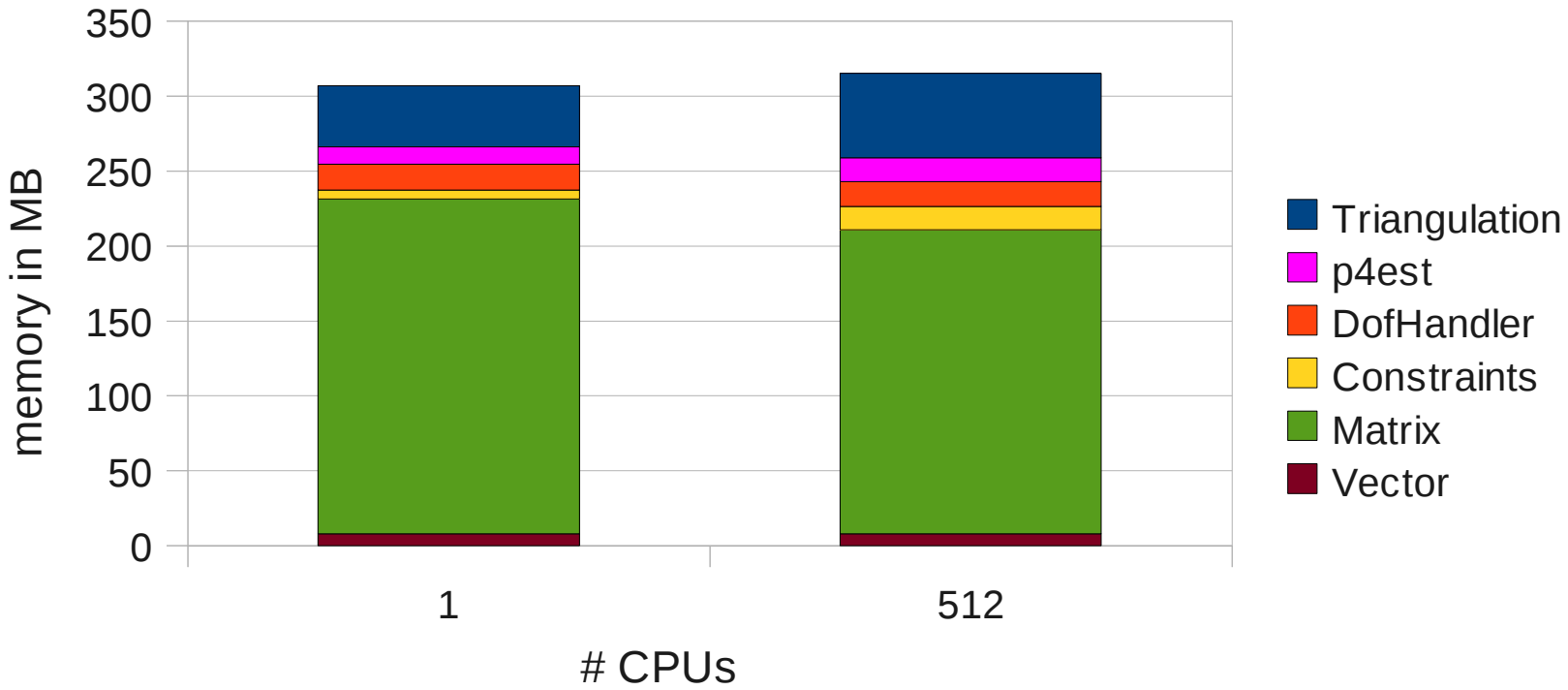
- Test scalability
- fix some parts that do not scale at the moment (involving all-to-all or vector<bool> for DoFs)
- More interesting test problems
- Switch to matrix-free (transfer, smoothing) with Martin's help
- Hybrid parallel experiments
- New assembling technique in deal.II?
- Run on Xeon Phi's?

Thanks for your attention!

Hybrid

- hybrid = MPI between nodes, multithreading inside node
- Advantage: save memory (important in the future, see Guido's talk)
- Bottleneck in codes: Preconditioners
- Inside PETSc/Trilinos: preconditioners are not multithreaded
  $\rightsquigarrow$ not worth it today
- But we are ready: multithreading in assembly, etc.

Test: memory consumption



3D, memory usage per object, weak scaling

What is ASPECT?

ASPECT = **A**dvanced **S**olver for **P**roblems in **E**arth's **C**onvec**T**ion

- Modern numerical methods
- Open source, C++: `http://www.dealii.org/aspect/`
- Based on the finite element library deal.II
- Supported by CIG
- Main author

Bangerth and Heister.
ASPECT: Advanced Solver for Problems in Earth's ConvecTion, 2012.
http://www.dealii.org/aspect/.

Kronbichler, Heister and Bangerth.
High Accuracy Mantle Convection Simulation through Modern Numerical Methods.
Geophysical Journal International, 2012, 191, 12-29.