Parallel Linear Algebra in deal.II

Timo Heister, Clemson University
heister@clemson.edu

2013-08-20
4th deal.II workshop, College Station, TX

CLEMSON UNIVERSITY

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

CIG COMPUTATIONAL INFRASTRUCTURE for GEODYNAMICS

What is this talk about?

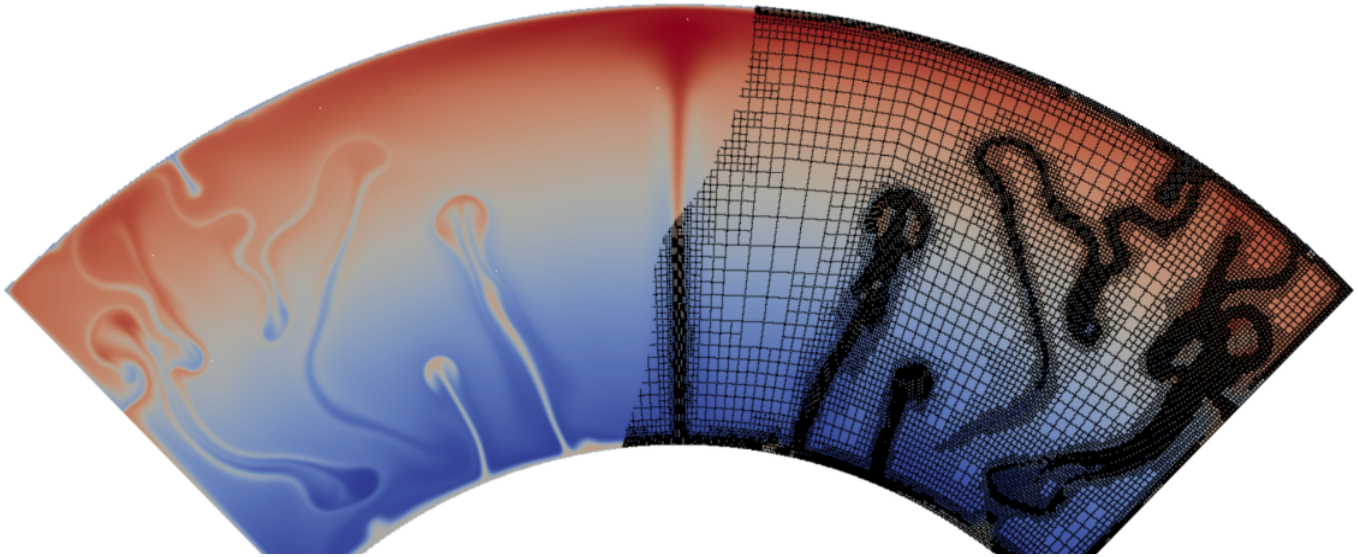
- Guide: parallel computations
- Explain recent changes in linear algebra
- Status and ideas for future work

Content:

1. How to parallelize?
2. Distributed meshes
3. Parallel linear algebra

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

Motivation



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

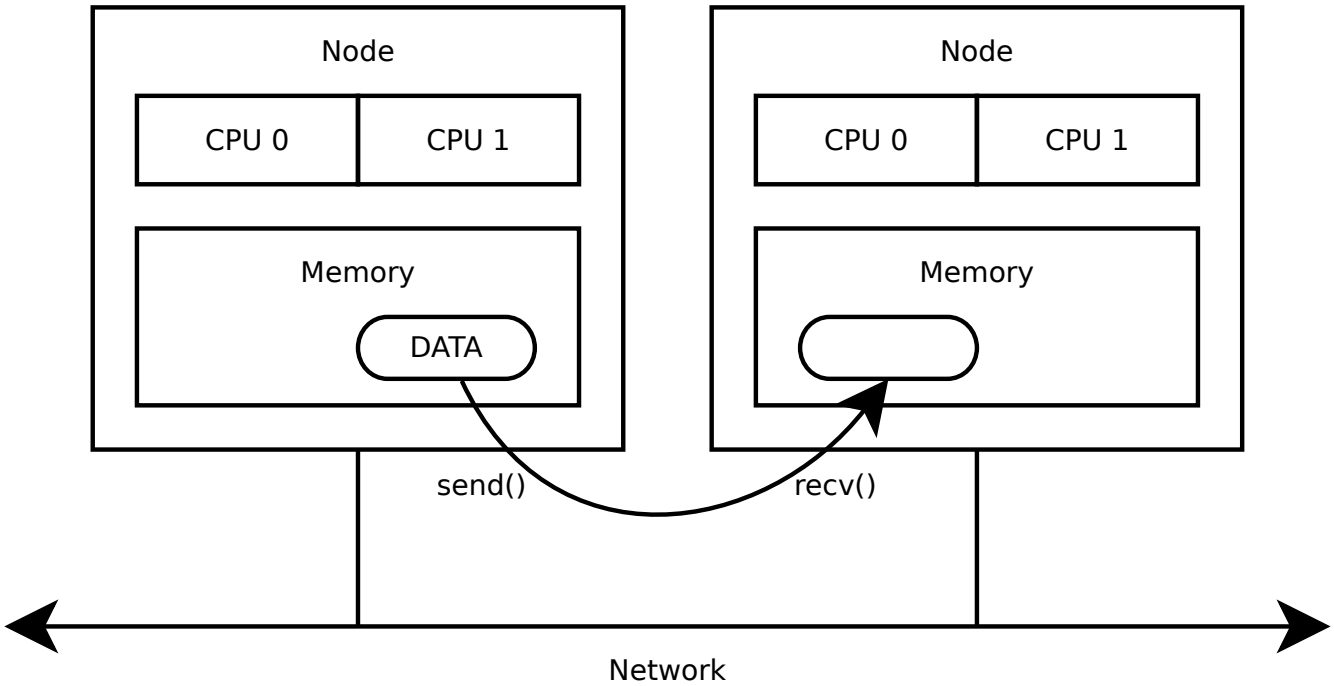
- ASPECT: `http://www.dealii.org/aspect/`
- Global mantle convection in the Earth's mantle
- 3d computations, adaptive meshes, 100 million+ DoFs
- Need: fast refinement, partitioning

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

Parallel Computing Model

- System:
  nodes connected via fast network
- Model: MPI (+multithreading)

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



How To?

- Required: split up the work!
- Goal: get solutions faster, allow larger problems
- Who needs this?
- Advice: $< 500'000$ DoFs and 2d is fine without parallel linear algebra, use a direct solver
  if not: listen!
- From laptop to supercomputer

Scalability

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

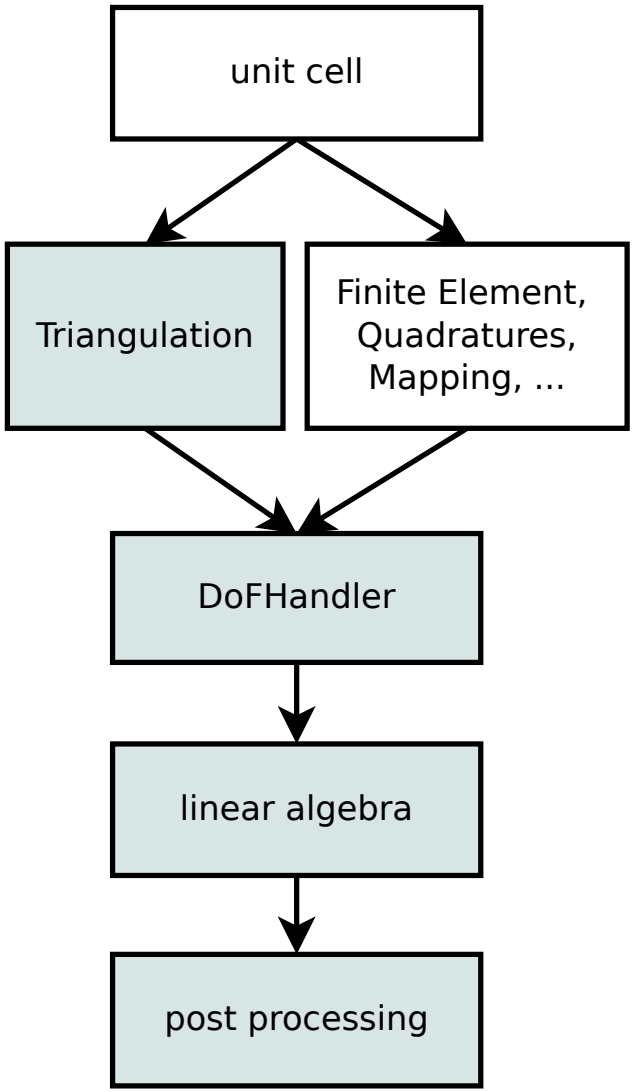
Required for Scalability:

- 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

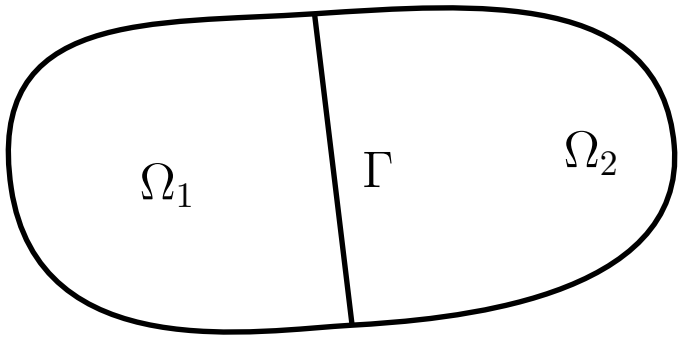
Needs to be parallelized:

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

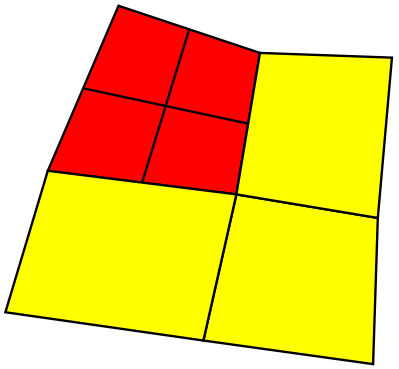


- Split up problem on PDE level
- Solve subproblems independently
- Converges against global solution
- Problems:
  - Boundary conditions are problem dependent:
    $\rightsquigarrow$ sometimes difficult!
    $\rightsquigarrow$ no black box approach!
  - Without coarse grid solver:
    condition number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

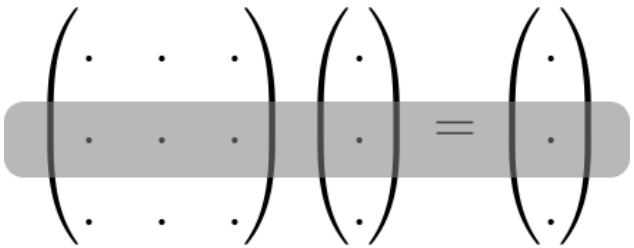
How to do Parallelization?

Option 2: Algebraic Splitting

- Split up mesh between processors:



- Assemble logically global linear system (distributed storage):

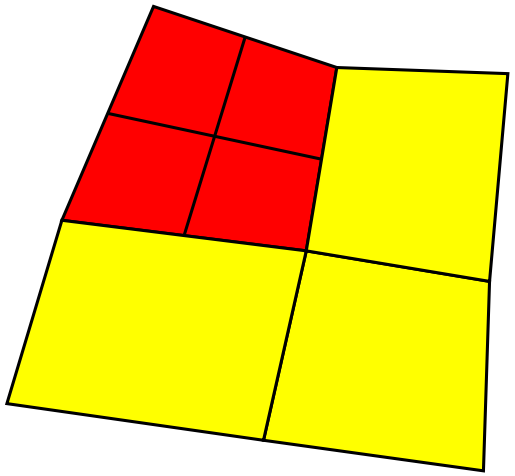


- Solve using iterative linear solvers in parallel
- Advantages:
  - Looks like serial program to the user
  - Linear scaling possible (with good preconditioner)

Partitioning

Optimal partitioning (coloring of cells):

- same size per region
  ⇝ even distribution of work
- minimize interface between region
  ⇝ reduce communication



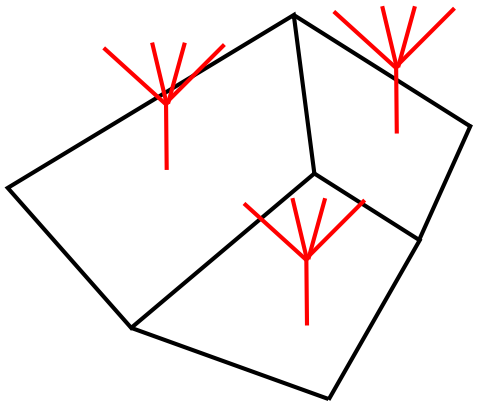
Optimal partitioning is an NP-hard graph partitioning problem.

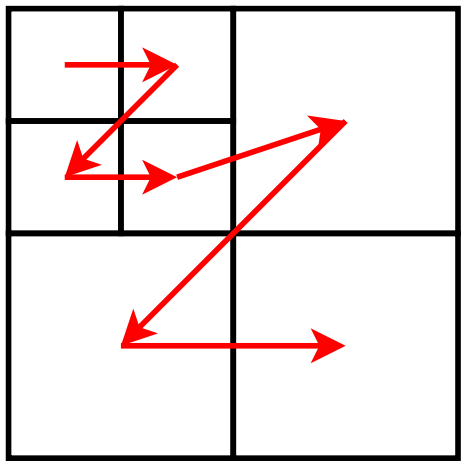
- Typically done: heuristics (existing tools: METIS)
- Problem: worse than linear runtime
- Large graphs: several minutes, memory restrictions

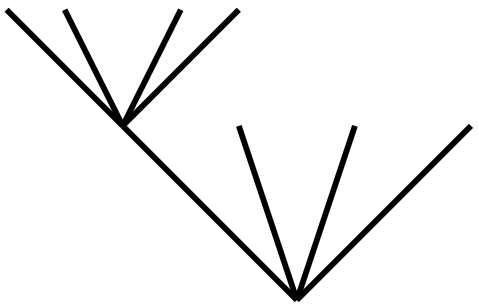
⇝ Alternative: avoid graph partitioning

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



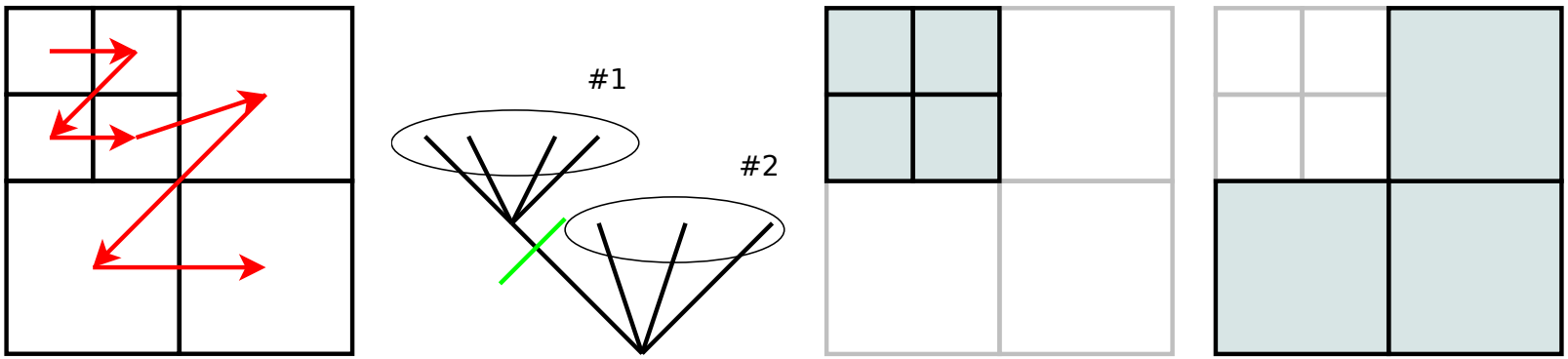




Burstedde, Wilcox, and Ghattas.
p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees.
SIAM J. Sci. Comput., 33 no. 3 (2011), pages 1103-1133.

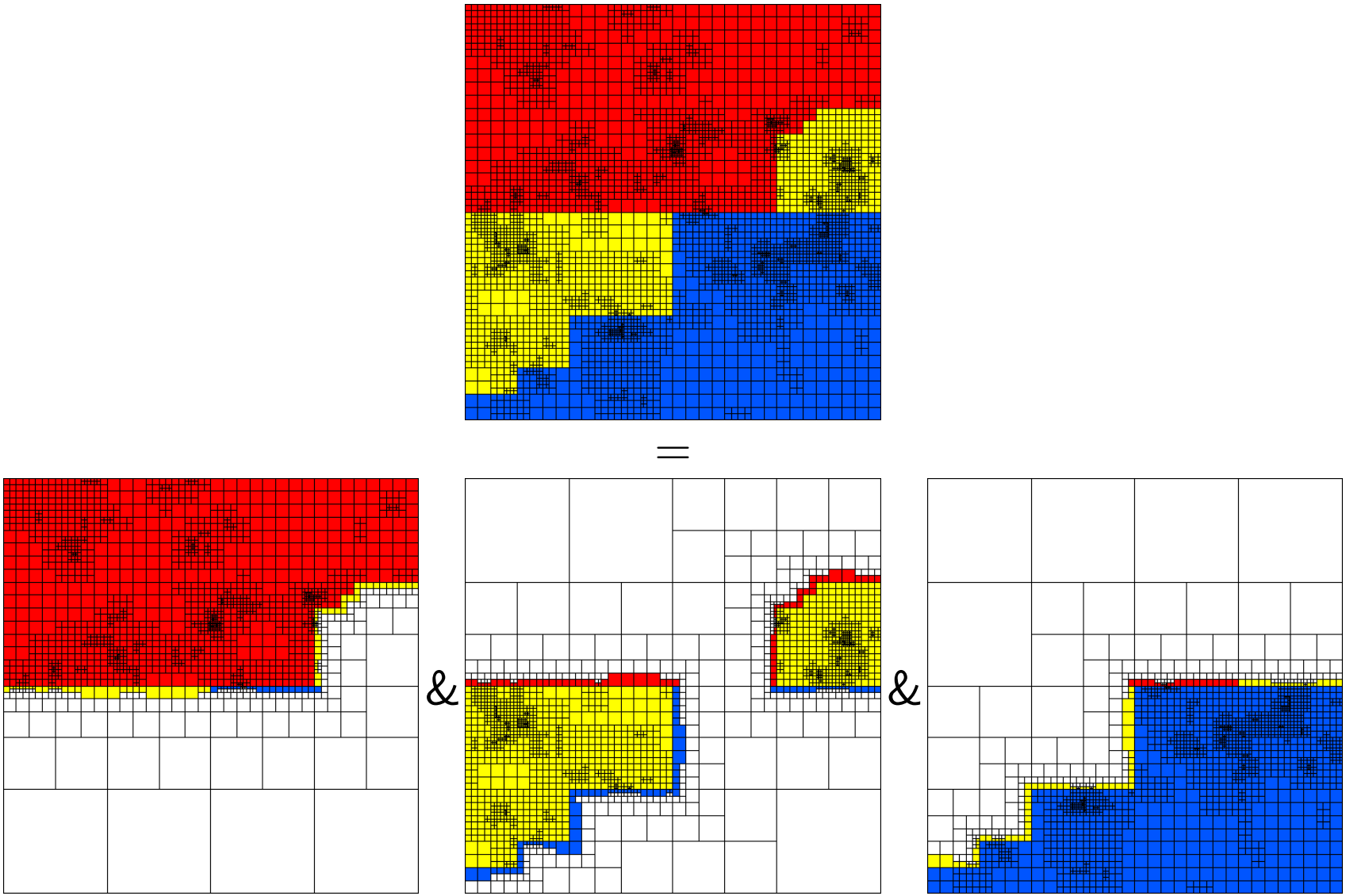
Triangulation

- 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)

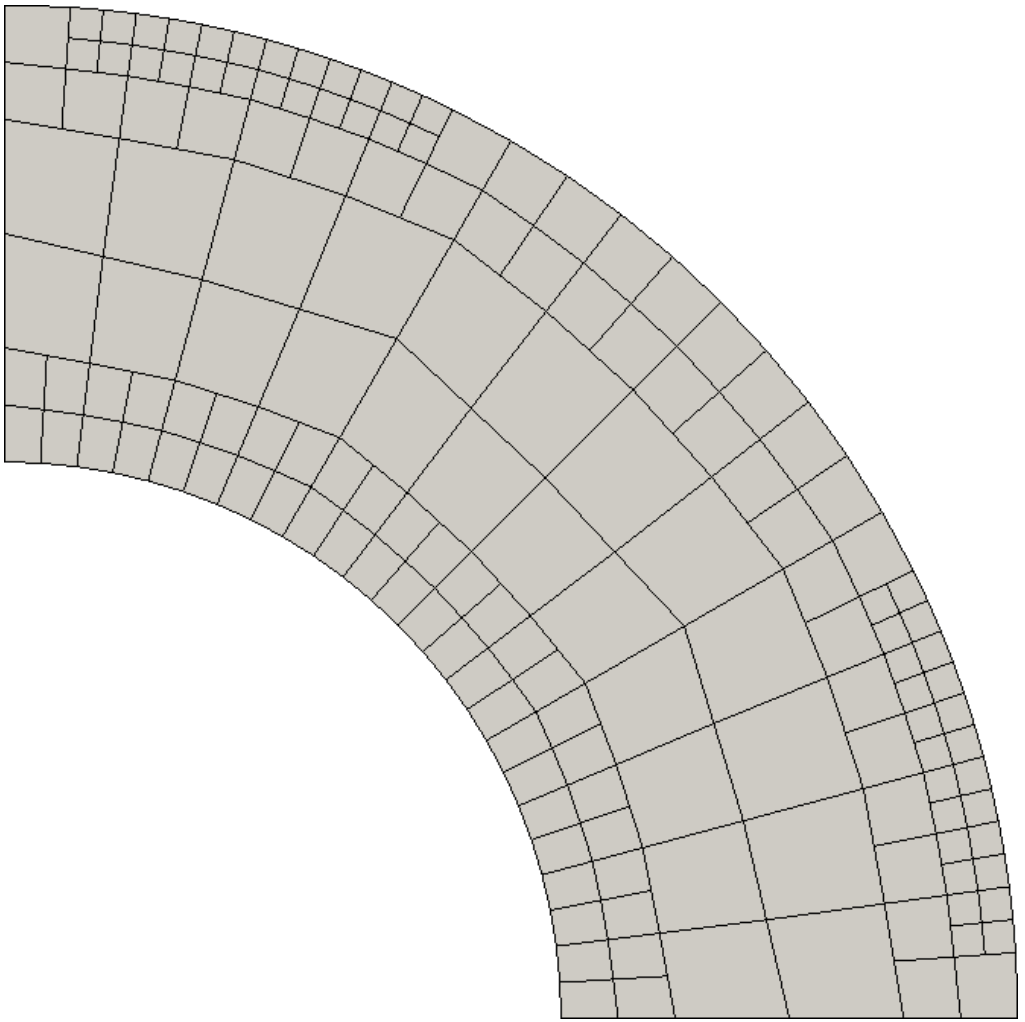
Example: Distributed Mesh Storage



Color: owned by CPU id

Arbitrary Geometry and Limitations

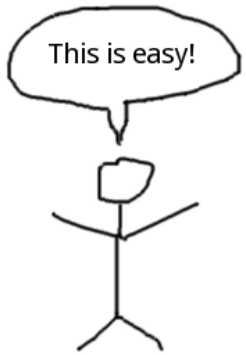
- Curved boundaries using higher order mappings
- Arbitrary geometry



Limitations:

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

In Practice



How to use?

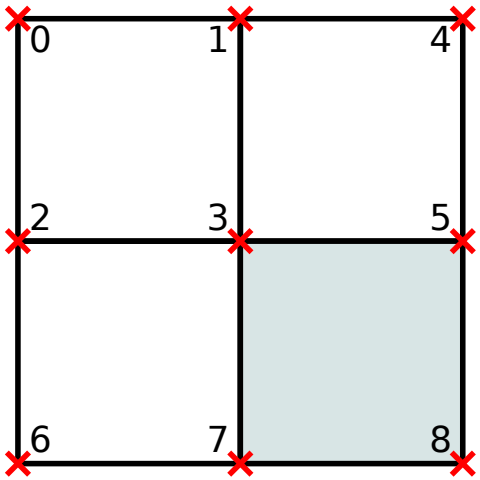
- Replace `Triangulation` by `parallel::distributed::Triangulation`
- Continue to load or create meshes as usual
- Adapt with `GridRefinement::refine_and_coarsen*` and `tr.execute_coarsening_and_refinement()`, etc.
- You can only look at own cells and ghost cells: `cell->is_locally_owned()`, `cell->is_ghost()`, 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	better?
hp?	yes	(planned)	yes?
geom. MG?	yes	in progress	?
Aniso. ref.?	yes	no	(offline only)
Periodicity	yes	in progress	?
Scalability	100 cores	16k+ cores	?

Distributing the Degrees of Freedom (DoFs)

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



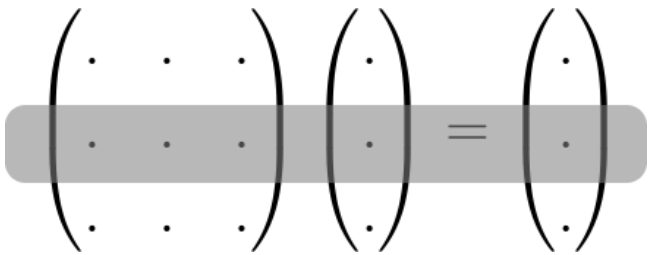
Sketch:

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

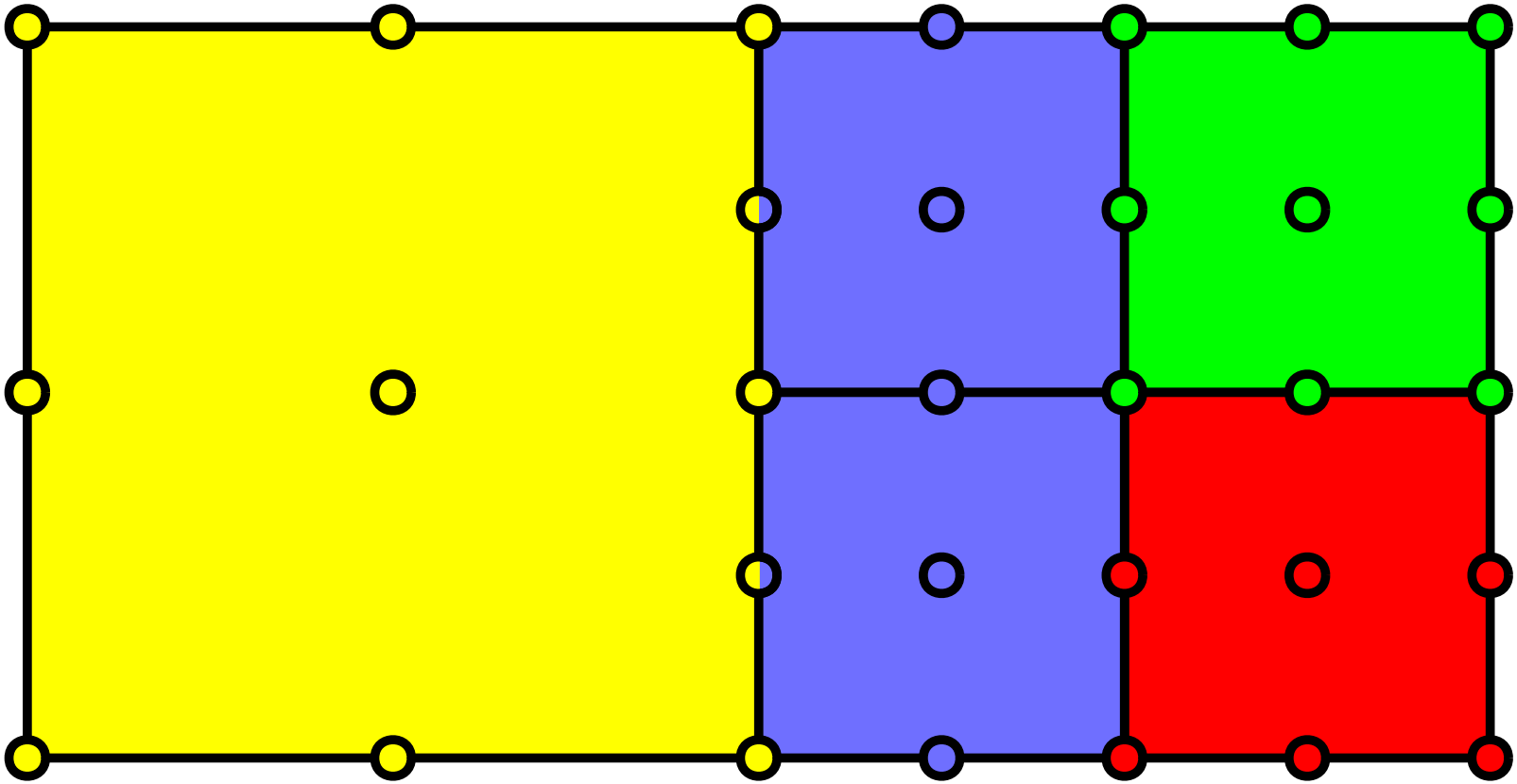
Linear Algebra: Short Version

- Want to go matrix-free?
  $\rightsquigarrow$ use `parallel::distributed::Vector`/etc., look at step-48, and talk to Martin Kronbichler

- This is what I normally tell:
  - Use distributed matrices and vectors (`PETScWrappers::Vector`/etc. or `TrilinosWrappers::Vector`/etc.)
  - Assemble local parts (some communication on interfaces)
  - Solve (preconditioners!)
  - Done.

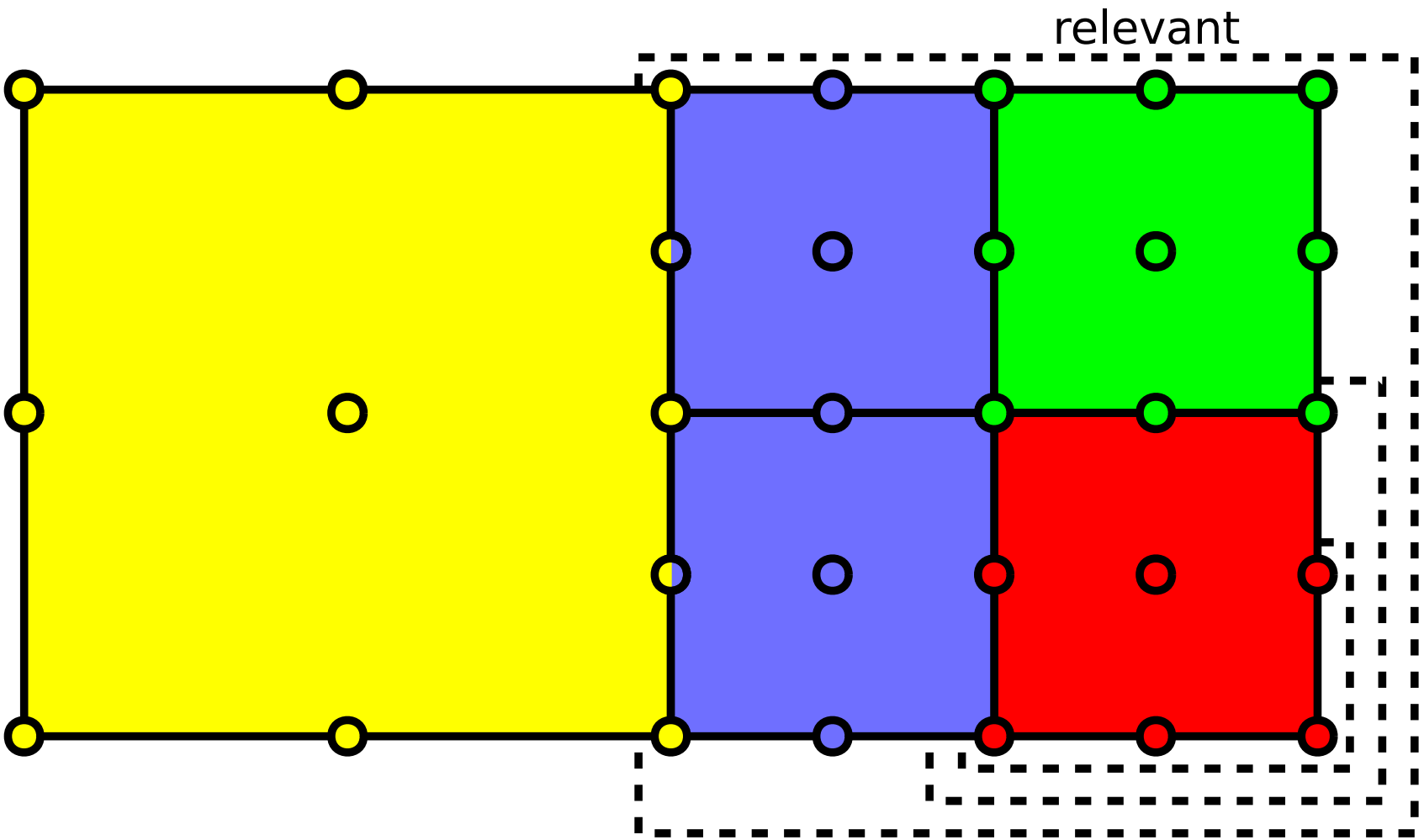


Longer Version



- 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 in between add/insert groups)
  - In no other case (all functions inside deal.II compress if necessary)! (this is new!)

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);

// ...
```

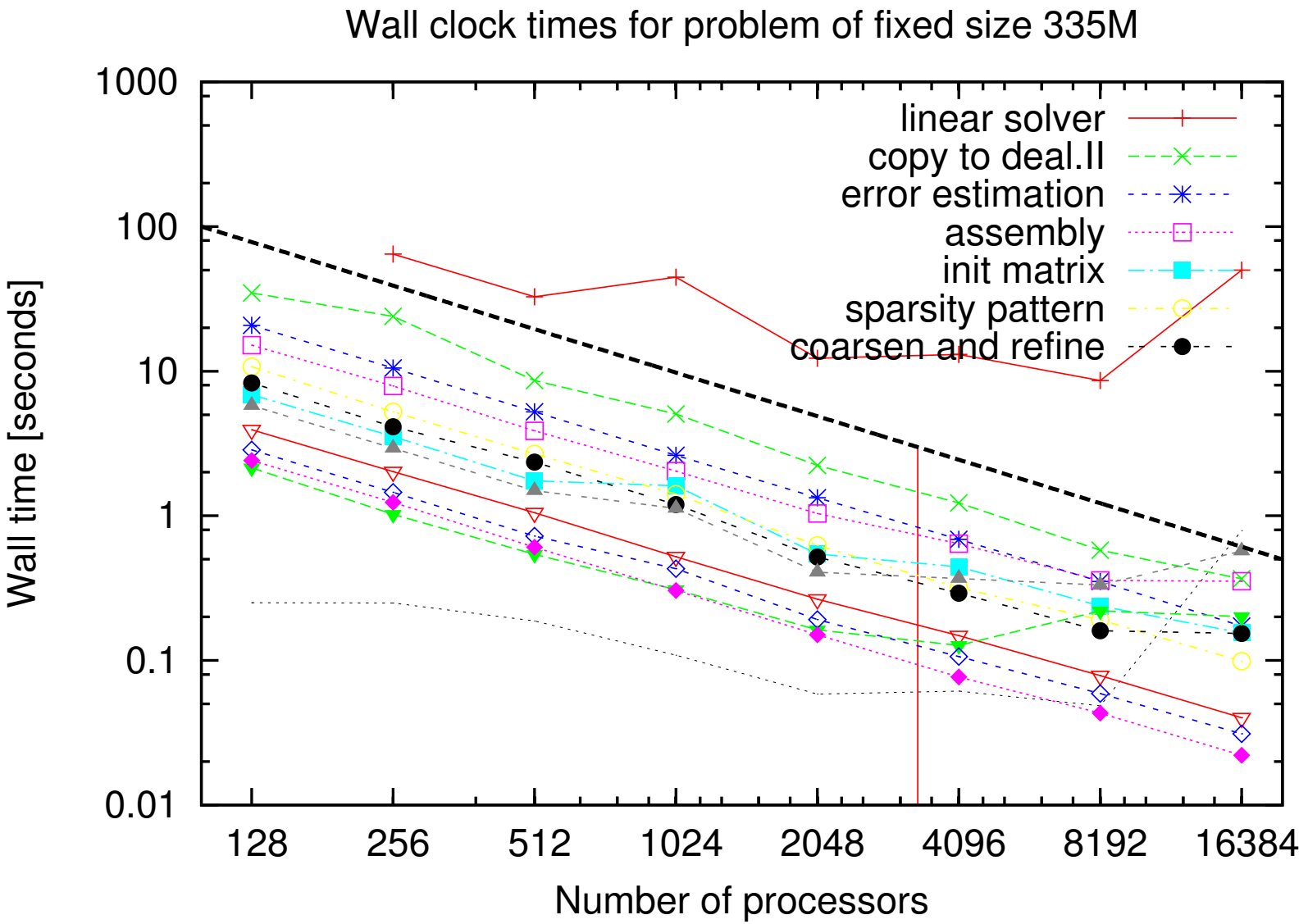
Linear Solvers

- Iterative solvers only need Mat-Vec products and scalar products
  $\rightsquigarrow$ equivalent to serial code
- Can use templated deal.II solvers like GMRES!
- Better: use tuned parallel iterative solvers that hide/minimize communication
- Preconditioners: more work, just operating on local blocks not enough

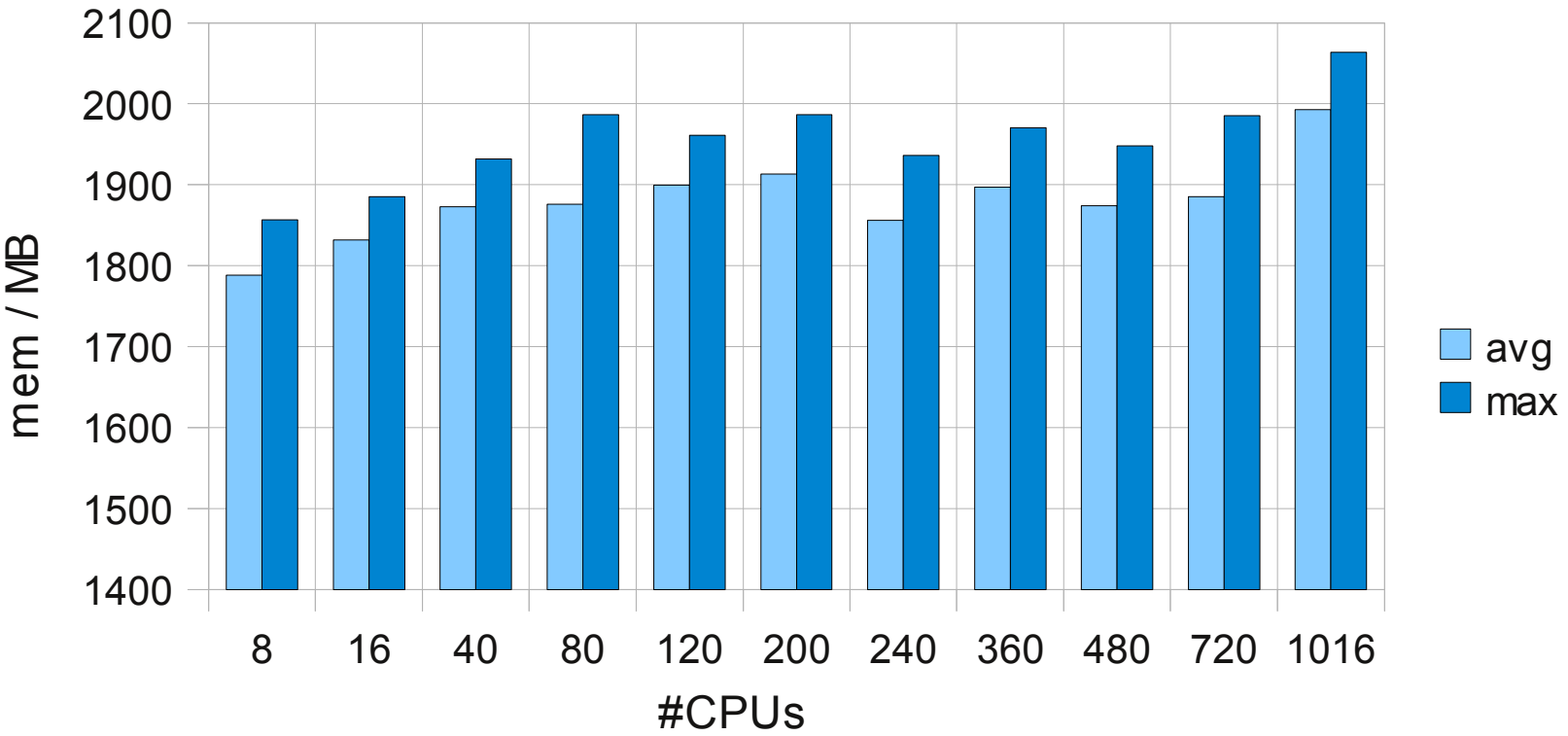
Hybrid

- 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.

Strong Scaling: 2d Adaptive Poisson Problem



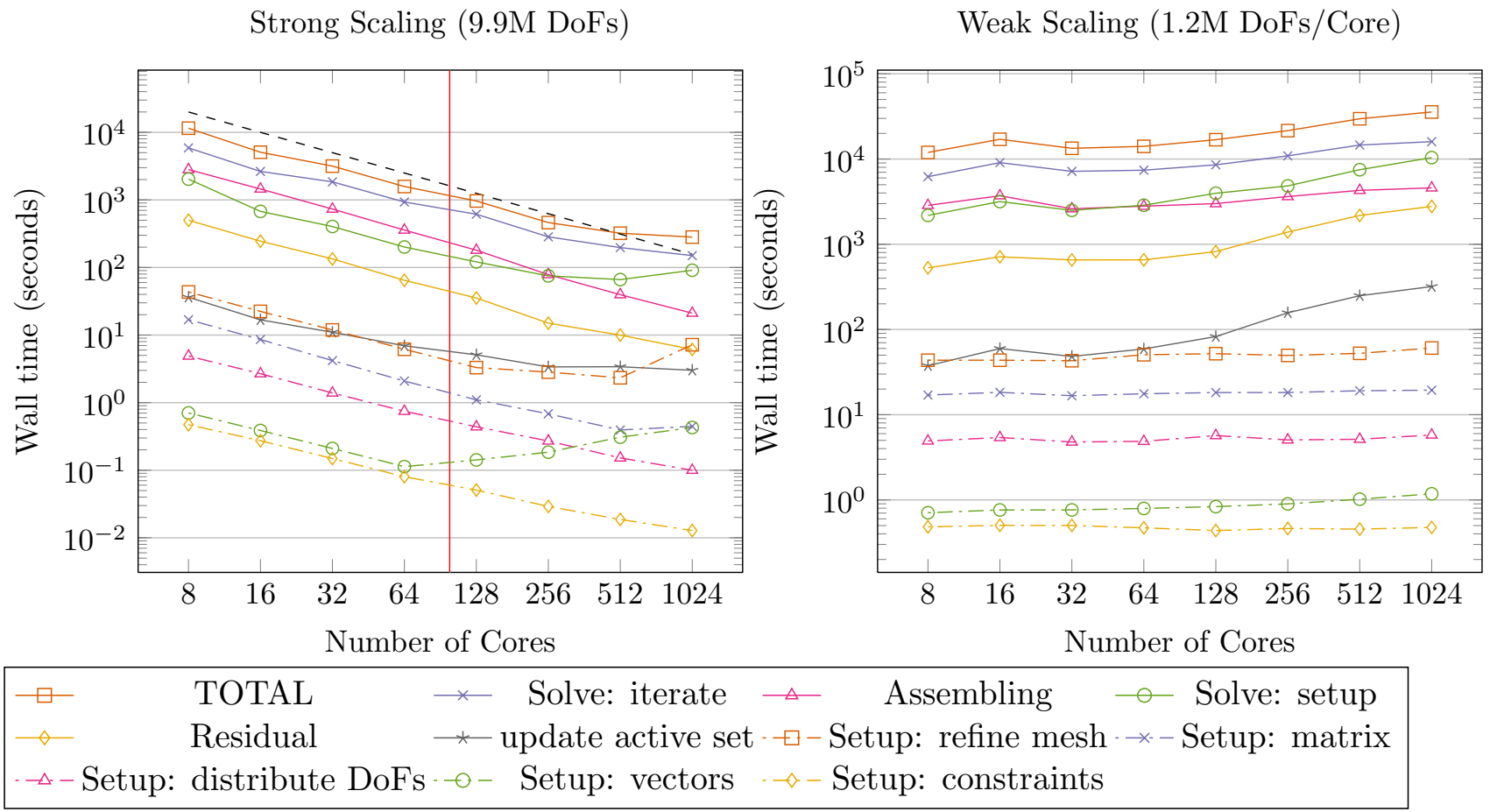
Test: Memory Consumption



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**

step-42: Plasticity



Postprocessing, . . .

Not covered today:

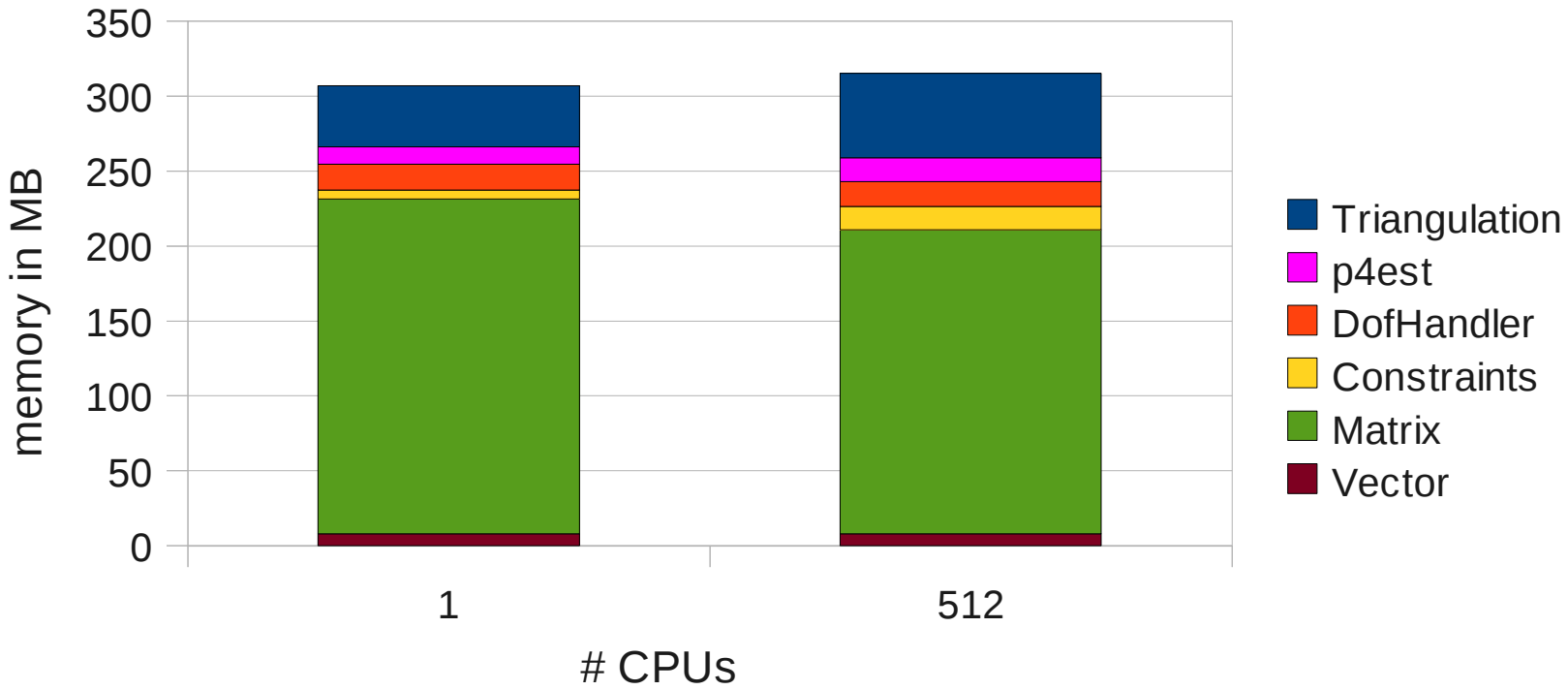
- Error estimation
- Decide over refinement and coarsening (communication!)
- Handling hanging nodes and other constraints
- Solution transfer (after refinement and repartitioning)
- Parallel I/O

Ideas:

- Geometric multigrid (parallel, adaptive) together with Guido
- Hp in parallel (any takers?)
- Old Tria: implement IndexSets
- Hybrid parallelization, petascale, accelerators?

Thanks for your attention!

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.