Master in High Performance Computing

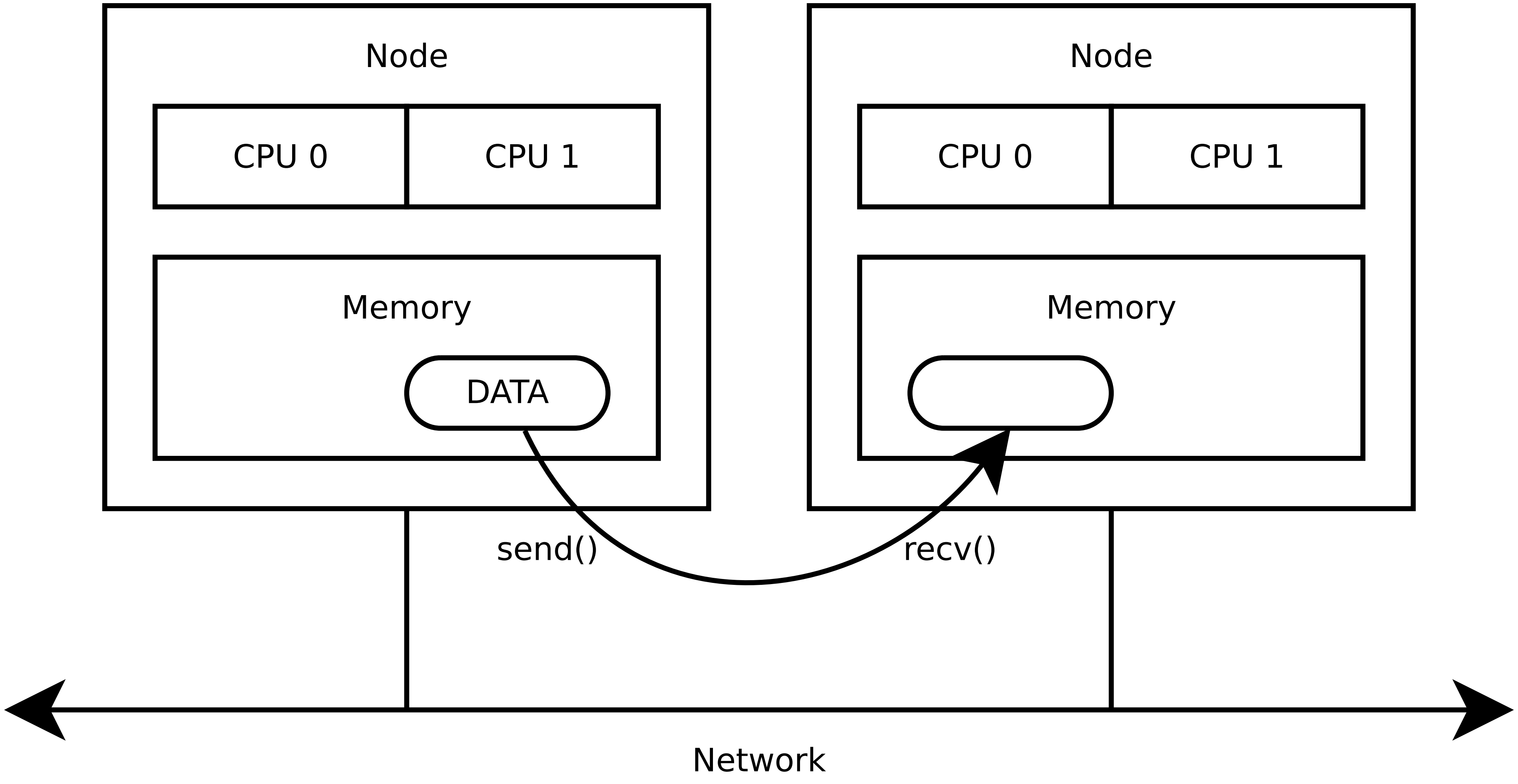
MPI Parallelization, Part I
parallel::distributed::Triangulation

Denis Davydov (denis.davydov@fau.de)
Luca Heltai (luca.heltai@sissa.it)

25 February - 1 March 2019

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Parallel computing model: MPI



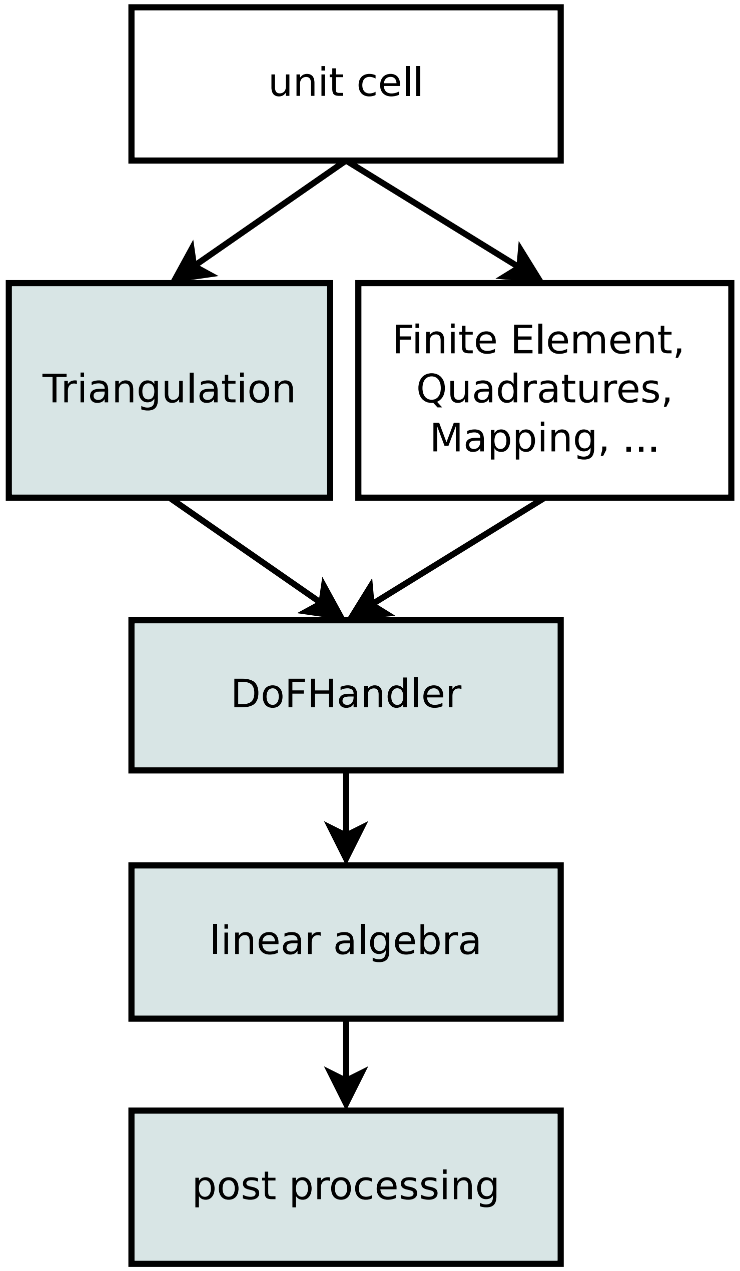
General Considerations

- Goal: get the solution faster!
- If FEM with <500.000 dofs, and 2d, use direct solver!

- If you need more, then you have to **SPLIT** the work
  - **Distributed data** storage everywhere
    - need special data structures
  - **Efficient algorithms**
    - not depending on total problem size
  - **“Localize” and “hide” communication**
    - point-to-point communication, nonblocking sends and receives

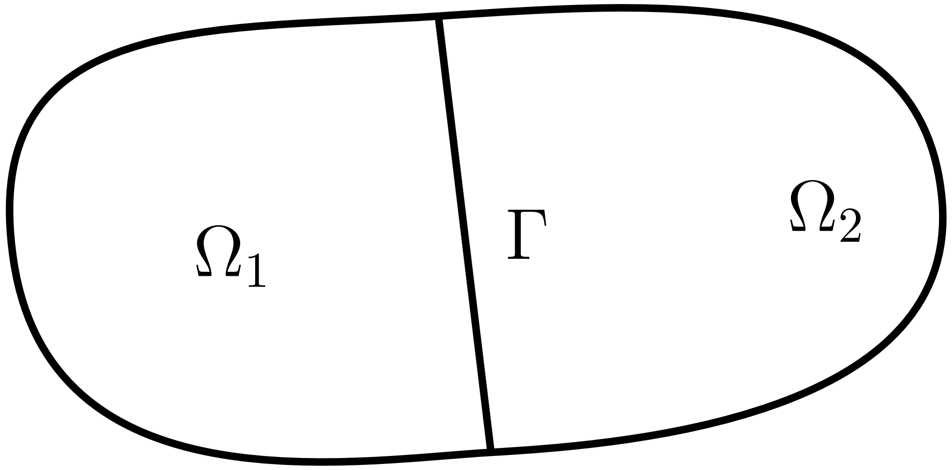
- Needs to be parallelized:

  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 Parallelize?

Option 1: Domain Decomposition

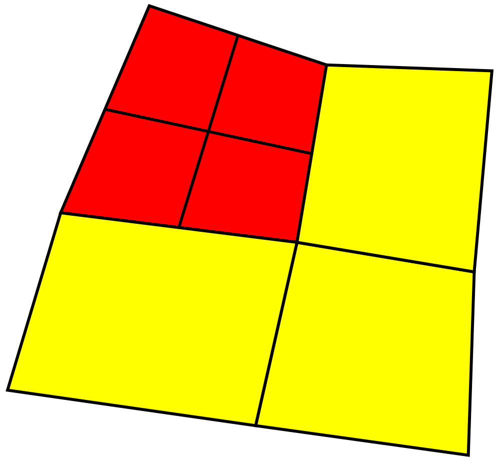


- Split up problem on PDE level
- Solve subproblems independently
- Converges against global solution
- Problems:
  - Boundary conditions are problem dependent:
    $\rightsquigarrow$ sometimes difficult!
    $\rightsquigarrow$ no black box approach!
  - Without coarse grid solver:
    condition number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

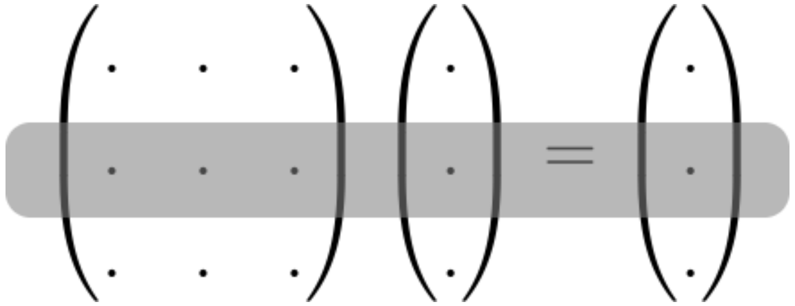
How to Parallelize?

Option 2: Algebraic Splitting

- Split up mesh between processors:



- Assemble logically global linear system (distributed storage):

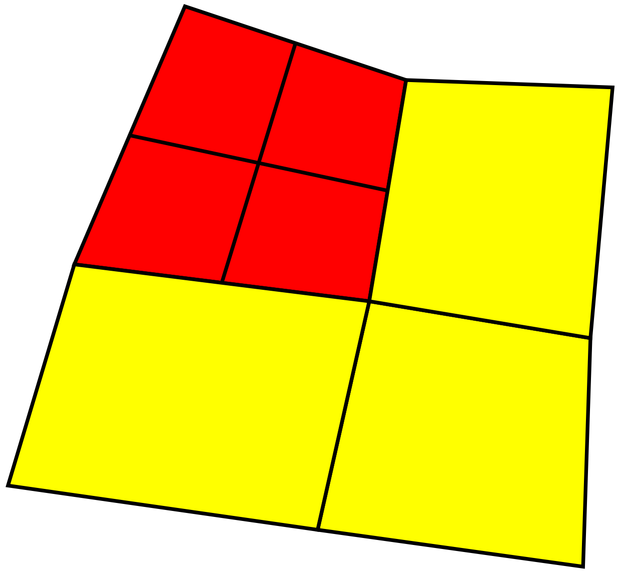


- Solve using iterative linear solvers in parallel
- Advantages:
  - Looks like serial program to the user
  - Linear scaling possible (with good preconditioner)

Partitioning

Optimal partitioning (coloring of cells):

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



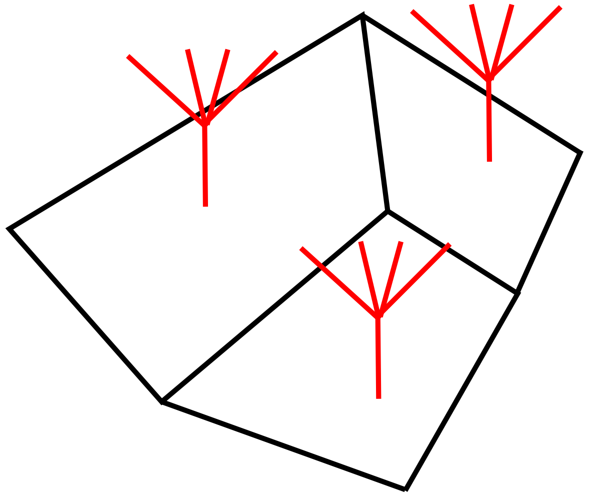
Optimal partitioning is an NP-hard graph partitioning problem.

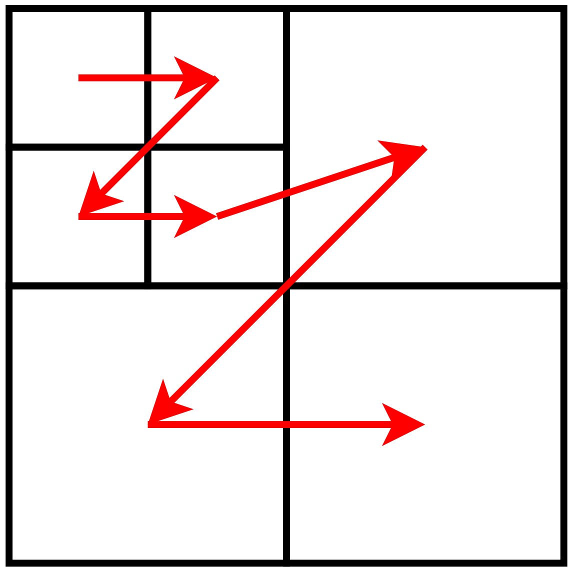
- Typically done: heuristics (existing tools: METIS)
- Problem: worse than linear runtime
- Large graphs: several minutes, memory restrictions

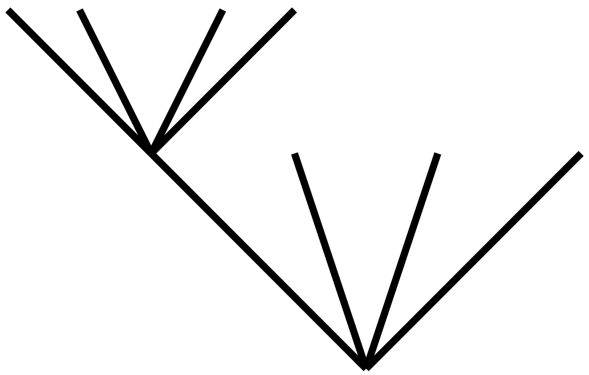
⇝ Alternative: avoid graph partitioning

Partitioning with “Space filling curves”

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



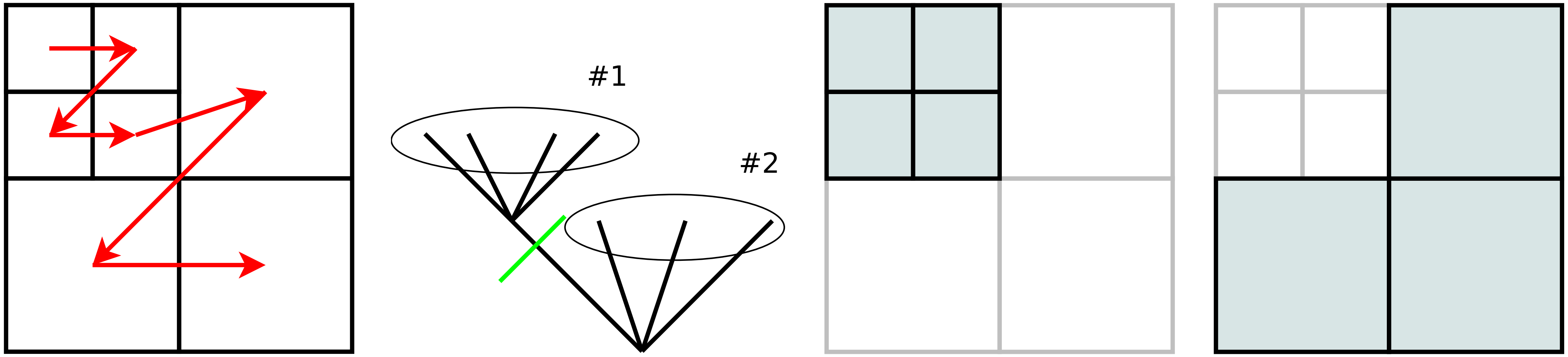




Burstedde, Wilcox, and Ghattas.
p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees.
SIAM J. Sci. Comput., 33 no. 3 (2011), pages 1103-1133.

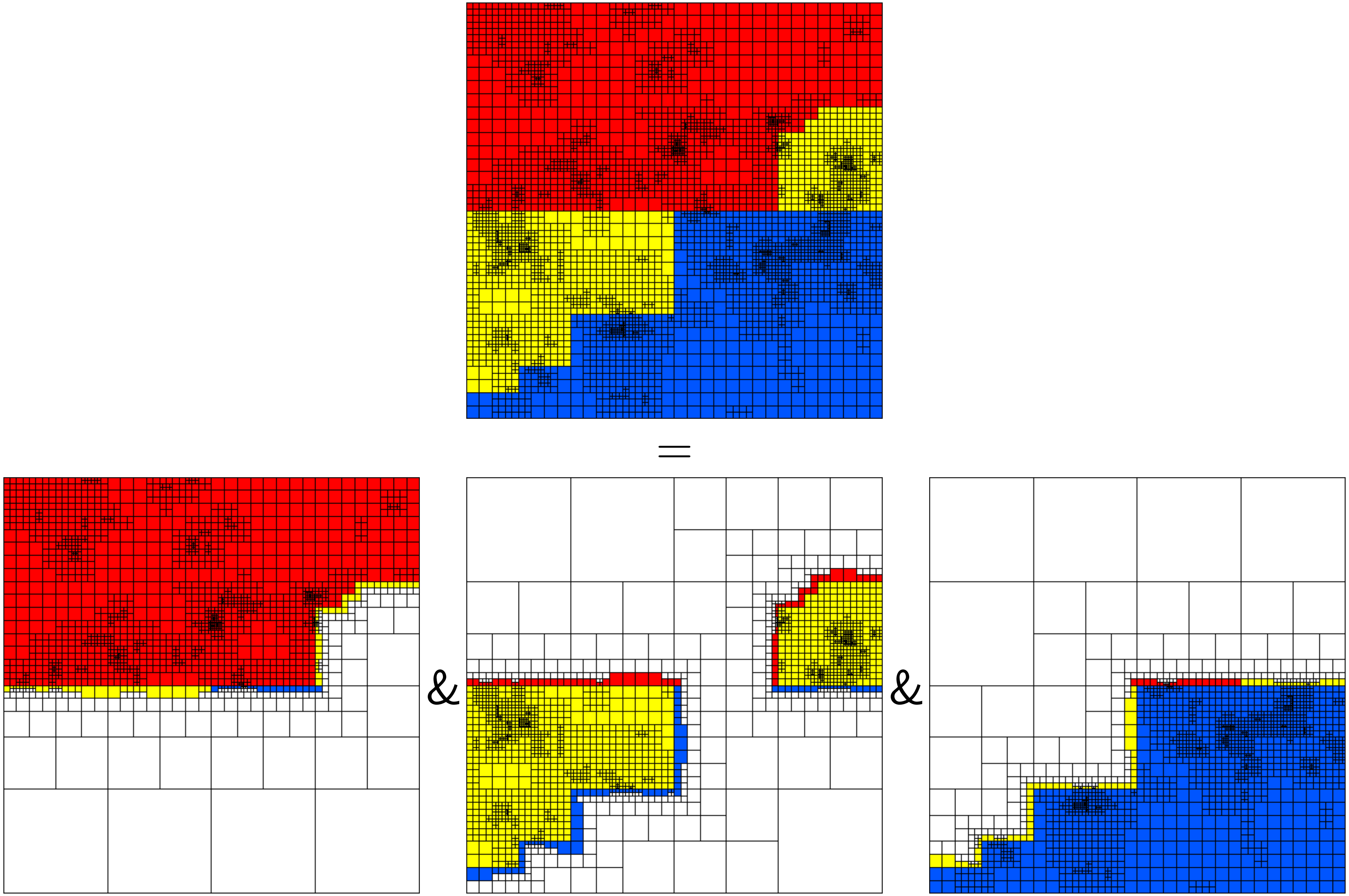
Triangulation

- 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 (color by CPU ID)



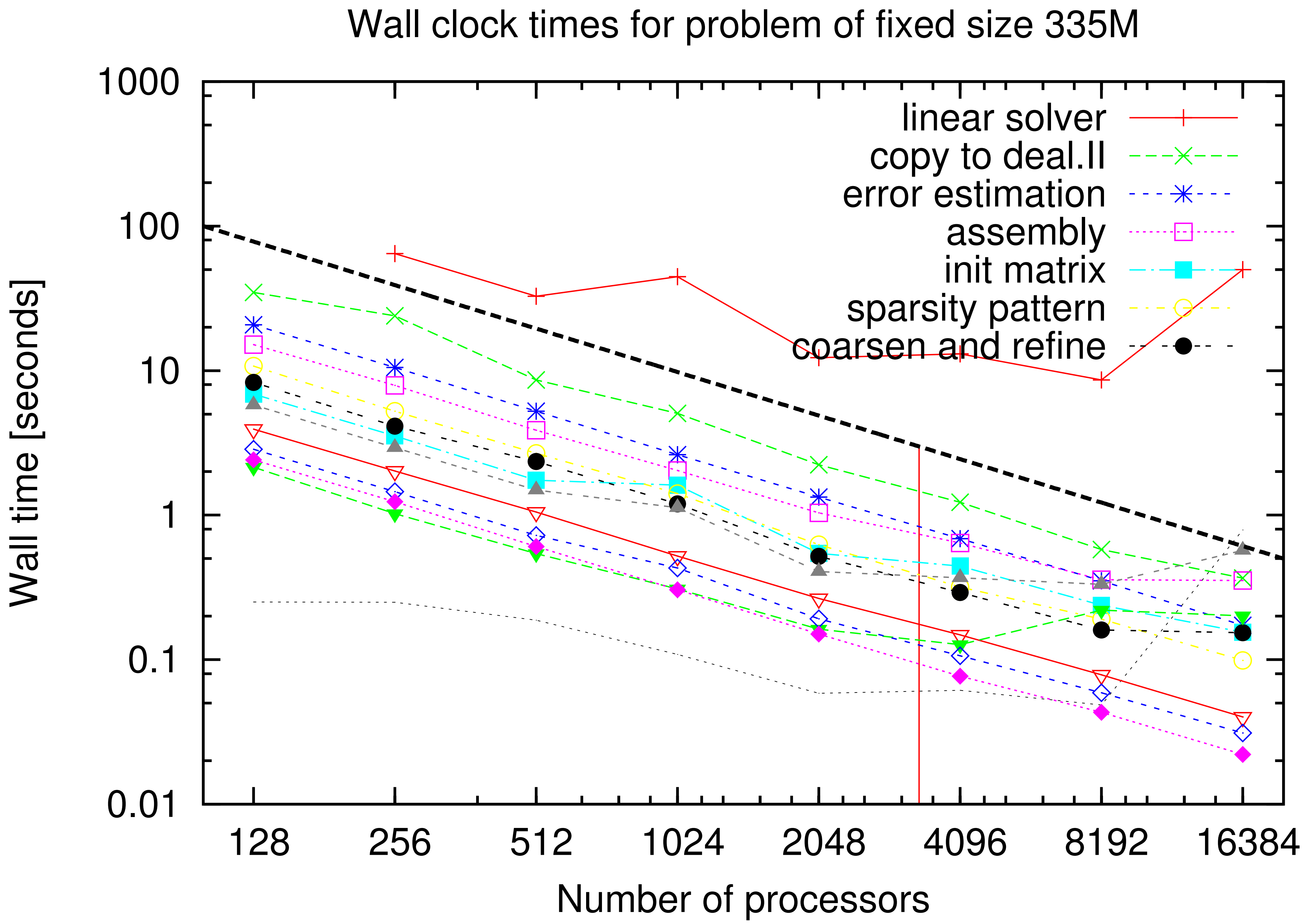
What’s needed?

	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	?

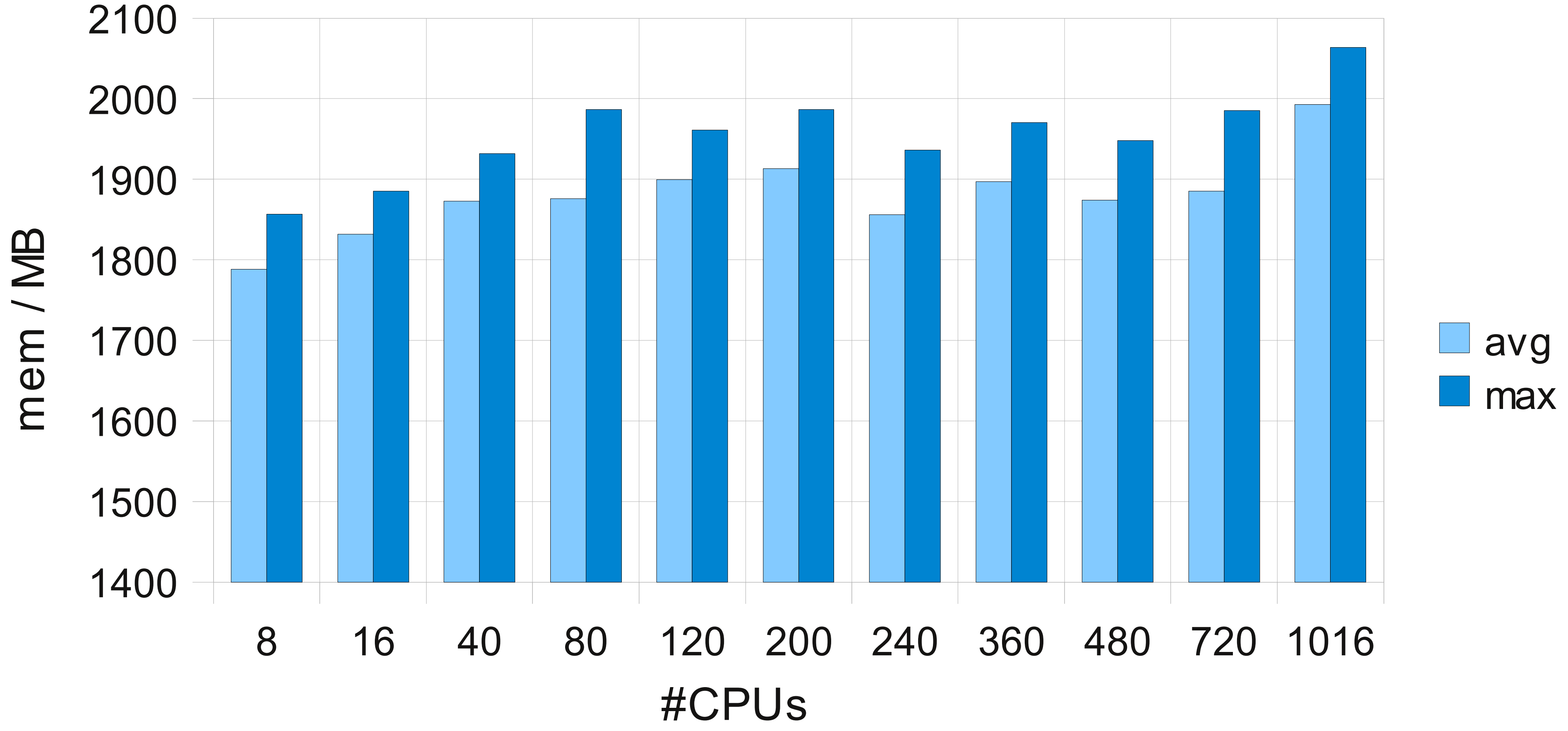
Solvers

- 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

Strong Scaling: 2d Poisson



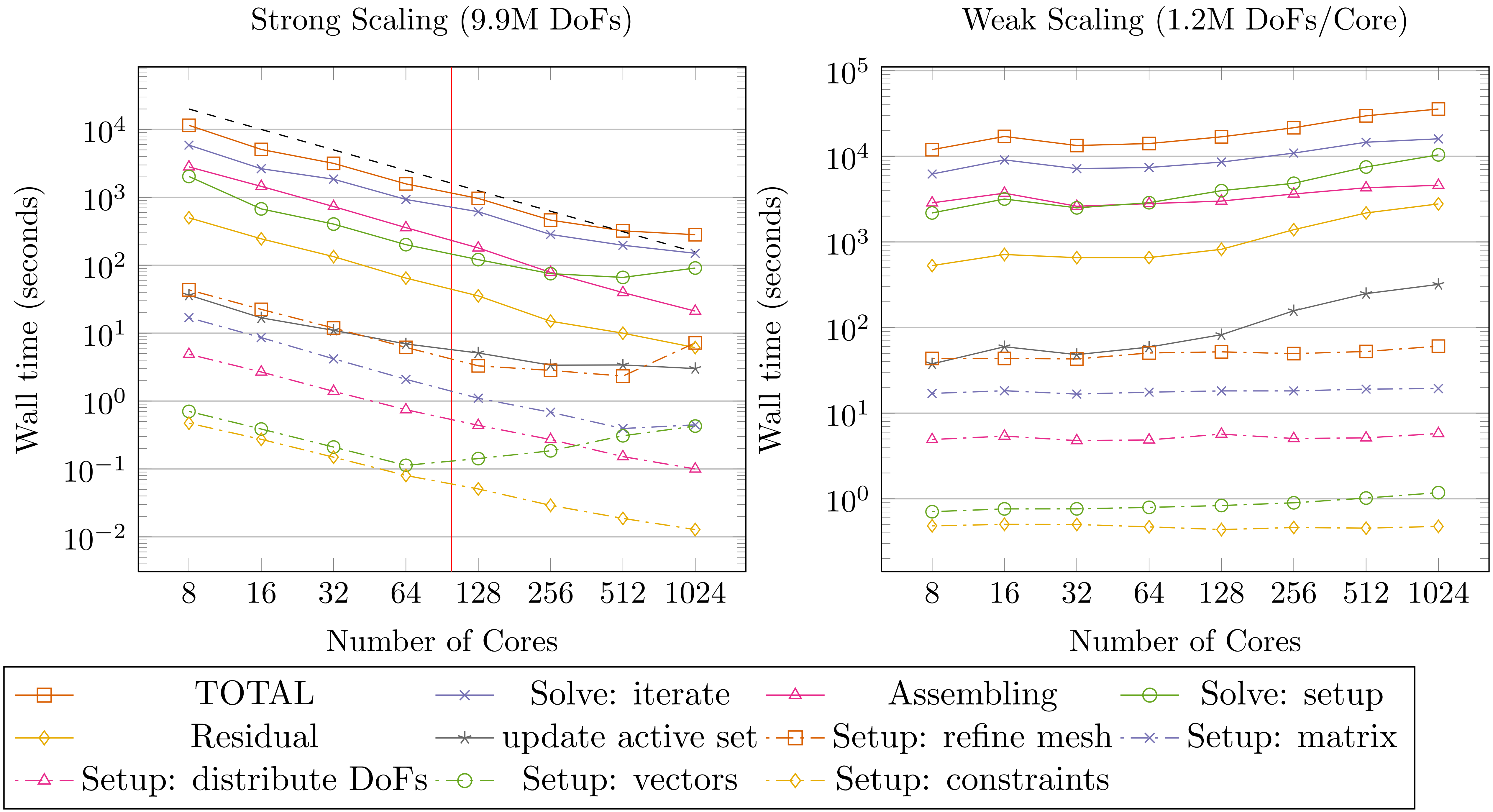
Memory Consumption



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)

⇝ **Constant memory usage with increasing # CPUs & problem size**

Step 40


Strong Scaling (9.9M DoFs)
Weak Scaling (1.2M DoFs/Core)
Wall time (seconds)
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
8
16
32
64
128
256
512
1024
Number of Cores
TOTAL
Solve: iterate
Assembling
Solve: setup
Residual
update active set
Setup: refine mesh
Setup: matrix
Setup: distribute DoFs
Setup: vectors
Setup: constraints

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

Trilinos VS PETSc

```cpp
#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);

// ...
```