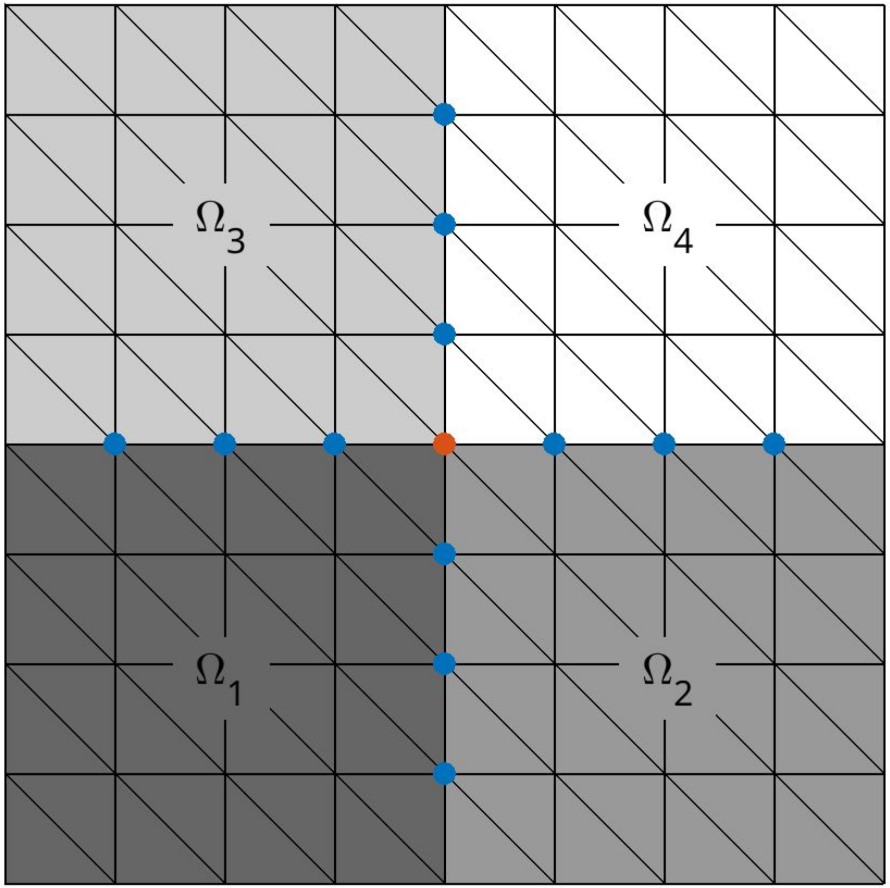
Karlsruhe Institute of Technology

# BDDC preconditioning in Ginkgo - a status update

**Fritz Goebel**, Hartwig Anzt, Terry Cojean, Marcel Koch, Fatemeh Chegini

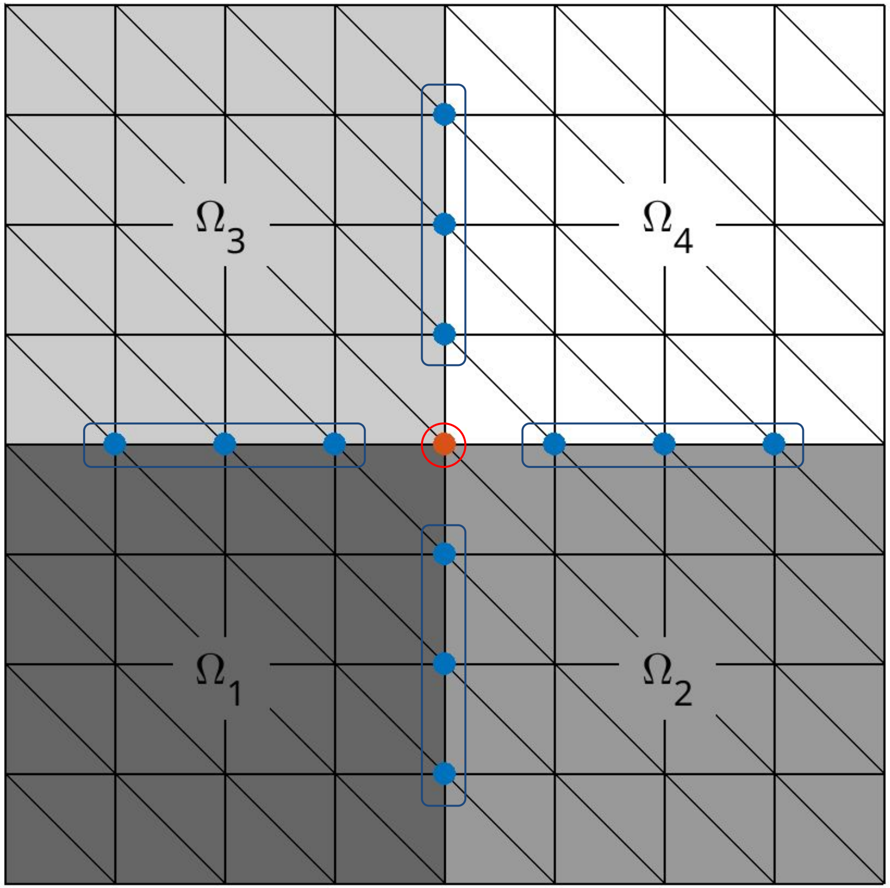
# Balancing Domain Decomposition by Constraints (BDDC)




- Consider the global Stiffness matrix

$$A = \sum_{i=1}^{N} R_i^T A_i R_i$$

# Balancing Domain Decomposition by Constraints (BDDC)




- Consider the global Stiffness matrix

$$A = \sum_{i=1}^{N} R_i^T A_i R_i$$

- Generate coarse system

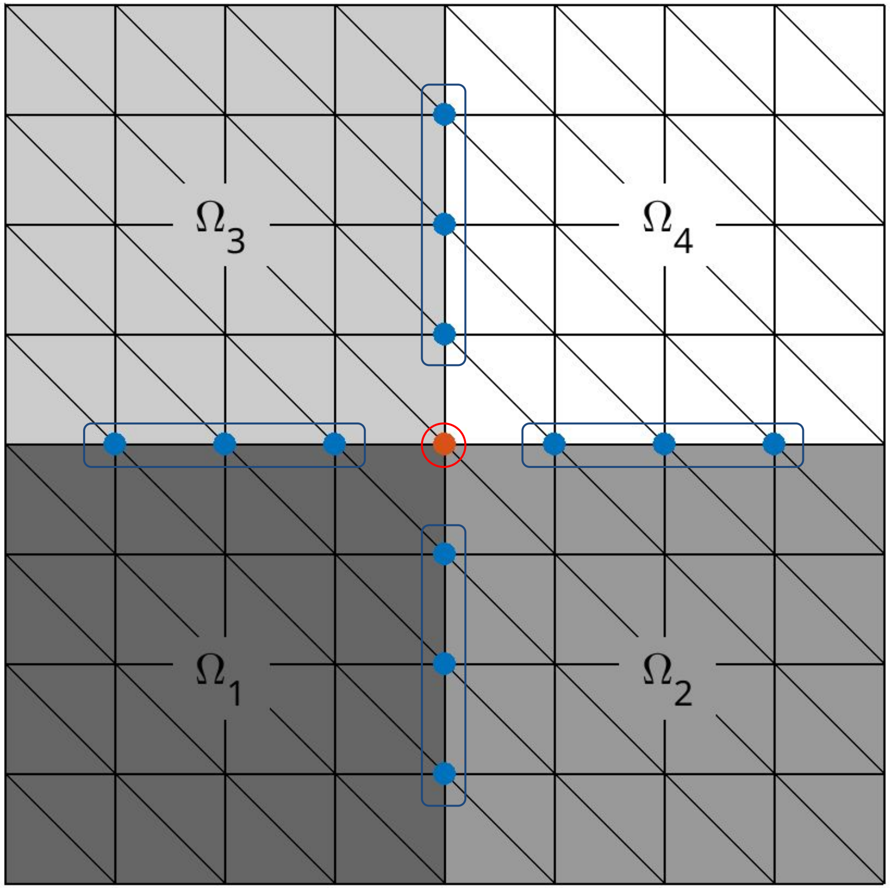
$$A_c = \sum_{i=1}^{N} R_{ci}^T A_{ci} R_{ci}$$

where

$$A_{ci} = \Phi_i^T A_i \Phi_i,$$

$$\begin{bmatrix} A_i & C_i^T \\ C_i & 0 \end{bmatrix} \begin{bmatrix} \Phi_i \\ \Lambda_i \end{bmatrix} = \begin{bmatrix} 0 \\ I \end{bmatrix}$$

# Balancing Domain Decomposition by Constraints (BDDC)




- Consider the global Stiffness matrix

$$A = \sum_{i=1}^{N} R_i^T A_i R_i$$

- Generate coarse system

$$A_c = \sum_{i=1}^{N} R_{ci}^T A_{ci} R_{ci}$$

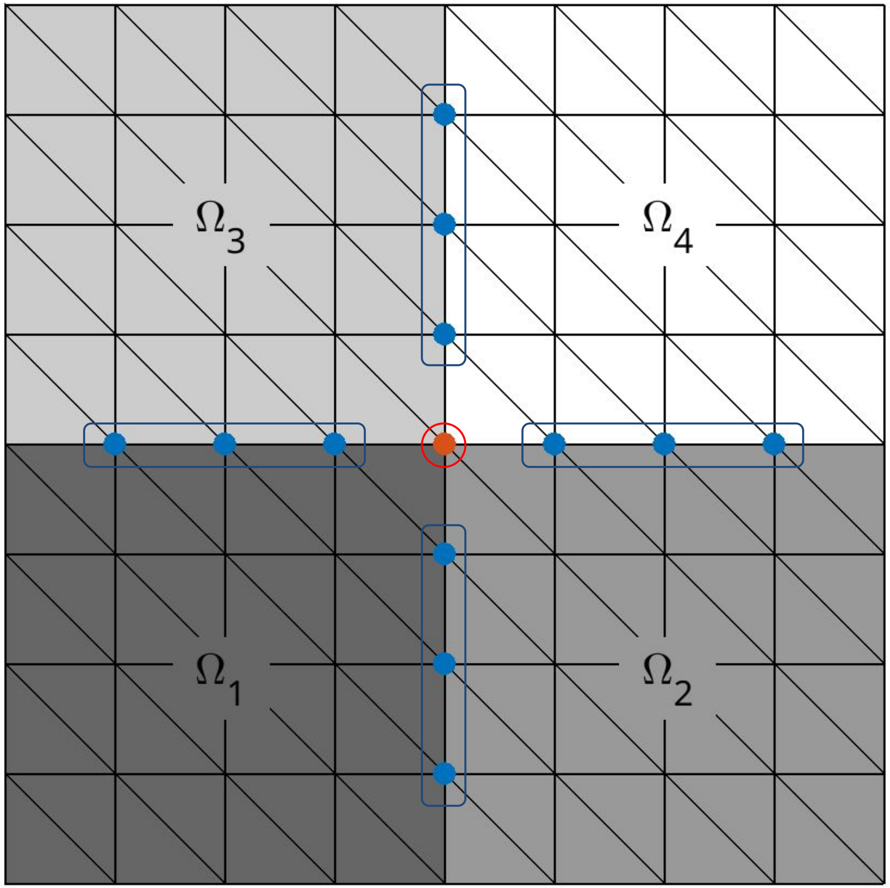
where

$$A_{ci} = \Phi_i^T A_i \Phi_i,$$

$$\begin{bmatrix} A_i & C_i^T \\ C_i & 0 \end{bmatrix} \begin{bmatrix} \Phi_i \\ \Lambda_i \end{bmatrix} = \begin{bmatrix} 0 \\ I \end{bmatrix}$$

Constraints

# Balancing Domain Decomposition by Constraints (BDDC)




- Consider the global Stiffness matrix

$$A = \sum_{i=1}^{N} R_i^T A_i R_i$$

- Generate coarse system

$$A_c = \sum_{i=1}^{N} R_{ci}^T A_{ci} R_{ci}$$

where

$$A_{ci} = \Phi_i^T A_i \Phi_i,$$

$$\begin{bmatrix} A_i & C_i^T \\ C_i & 0 \end{bmatrix} \begin{bmatrix} \Phi_i \\ \Lambda_i \end{bmatrix} = \begin{bmatrix} 0 \\ I \end{bmatrix}$$

Constraints

Full values on corners
Averages over interfaces

# Balancing Domain Decomposition by Constraints (BDDC)

The preconditioned residual consists of three parts:

$$M^{-1} = v_1 + v_2 + v_3$$

# Balancing Domain Decomposition by Constraints (BDDC)

The preconditioned residual consists of three parts:

$$M^{-1} = v_1 + v_2 + v_3$$

Coarse grid correction

$$v_1 = \sum_{i=1}^{N} R_i^T W_i \Phi_i R_{ci} A_c^{-1} r_c$$

# Balancing Domain Decomposition by Constraints (BDDC)

The preconditioned residual consists of three parts:

$$M^{-1} = v_1 + v_2 + v_3$$

Coarse grid correction

$$v_1 = \sum_{i=1}^{N} R_i^T W_i \Phi_i R_{ci} A_c^{-1} r_c$$

weights: $\frac{1}{\#\text{subdomains sharing dof}}$

Computed as described in Dohrmann

# Balancing Domain Decomposition by Constraints (BDDC)

The preconditioned residual consists of three parts:

$$M^{-1} = v_1 + v_2 + v_3$$

Coarse grid correction

$$v_1 = \sum_{i=1}^{N} R_i^T W_i \Phi_i R_{ci} A_c^{-1} r_c$$

$$r_c = R_{ci}^T \Phi_i^T W_i R_i r$$

## Balancing Domain Decomposition by Constraints (BDDC)

The preconditioned residual consists of three parts:

$$M^{-1} = v_1 + v_2 + v_3$$

Coarse grid correction

Subdomain correction

$$v_1 = \sum_{i=1}^{N} R_i^T W_i \Phi_i R_{ci} A_c^{-1} r_c$$

$$r_c = R_{ci}^T \Phi_i^T W_i R_i r$$

$$v_2 = \sum_{i=1}^{N} R_i^T W_i z_i$$

$$\begin{bmatrix} A_i & C_i^T \\ C_i & 0 \end{bmatrix} \begin{bmatrix} z_i \\ \lambda_i \end{bmatrix} = \begin{bmatrix} W_i R_i r \\ 0 \end{bmatrix}$$

Computed as described in Dohrmann

## Balancing Domain Decomposition by Constraints (BDDC)

The preconditioned residual consists of three parts:

$$M^{-1} = v_1 + v_2 + v_3$$

Coarse grid correction

$$v_1 = \sum_{i=1}^{N} R_i^T W_i \Phi_i R_{ci} A_c^{-1} r_c$$

$$r_c = R_{ci}^T \Phi_i^T W_i R_i r$$

Subdomain correction

$$v_2 = \sum_{i=1}^{N} R_i^T W_i z_i$$

$$\begin{bmatrix} A_i & C_i^T \\ C_i & 0 \end{bmatrix} \begin{bmatrix} z_i \\ \lambda_i \end{bmatrix} = \begin{bmatrix} W_i R_i r \\ 0 \end{bmatrix}$$

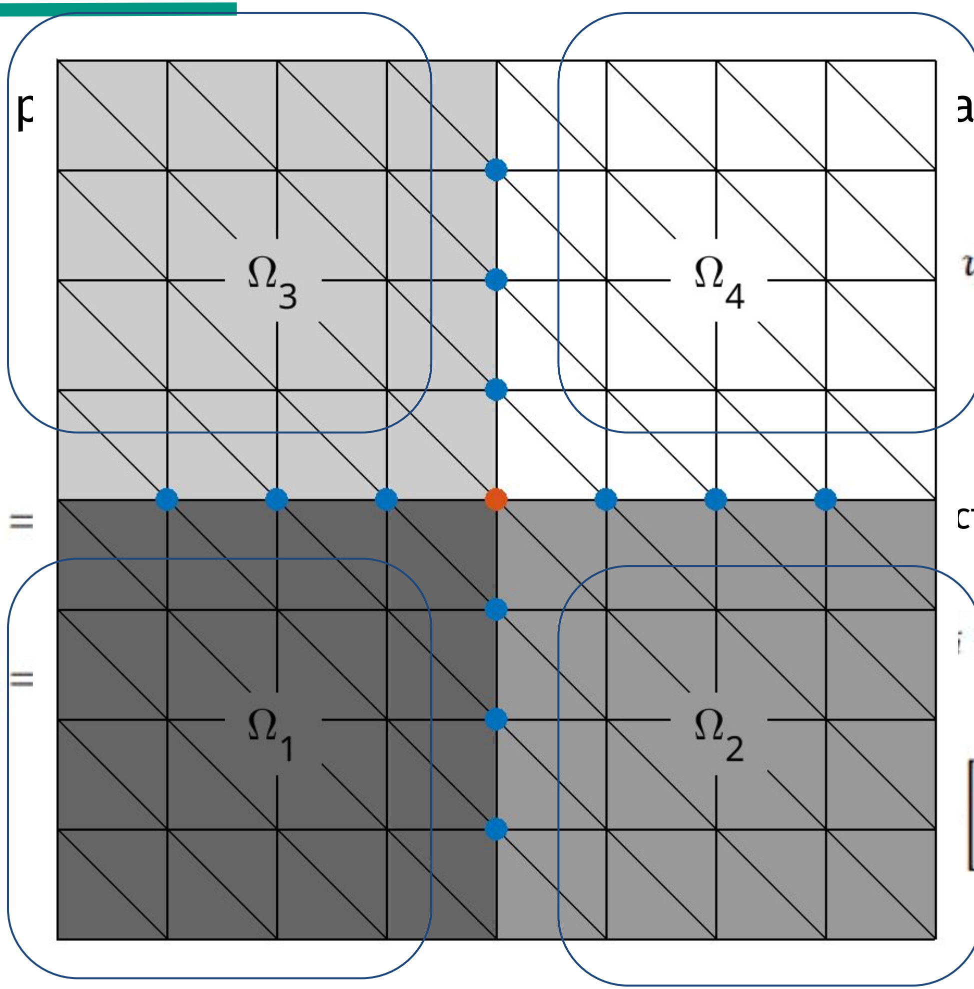
Static condensation correction

$$v_3 = \sum_{i=1}^{N} R_i^T R_{Ii}^T (R_{Ii} A_i R_{Ii}^T)^{-1} R_{Ii} R_i r_1$$

$$r_1 = r - A(v_1 + v_2)$$

# Balancing Domain Decomposition by Constraints (BDDC)

The p arts:




$v_3$

Static condensation correction

ction

$$v_3 = \sum_{i=1}^{N} R_i^T R_{Ii}^T (R_{Ii} A_i R_{Ii}^T)^{-1} R_{Ii} R_i r_1$$

$$r_1 = r - A(v_1 + v_2)$$

$v_1 =$

$r_c =$

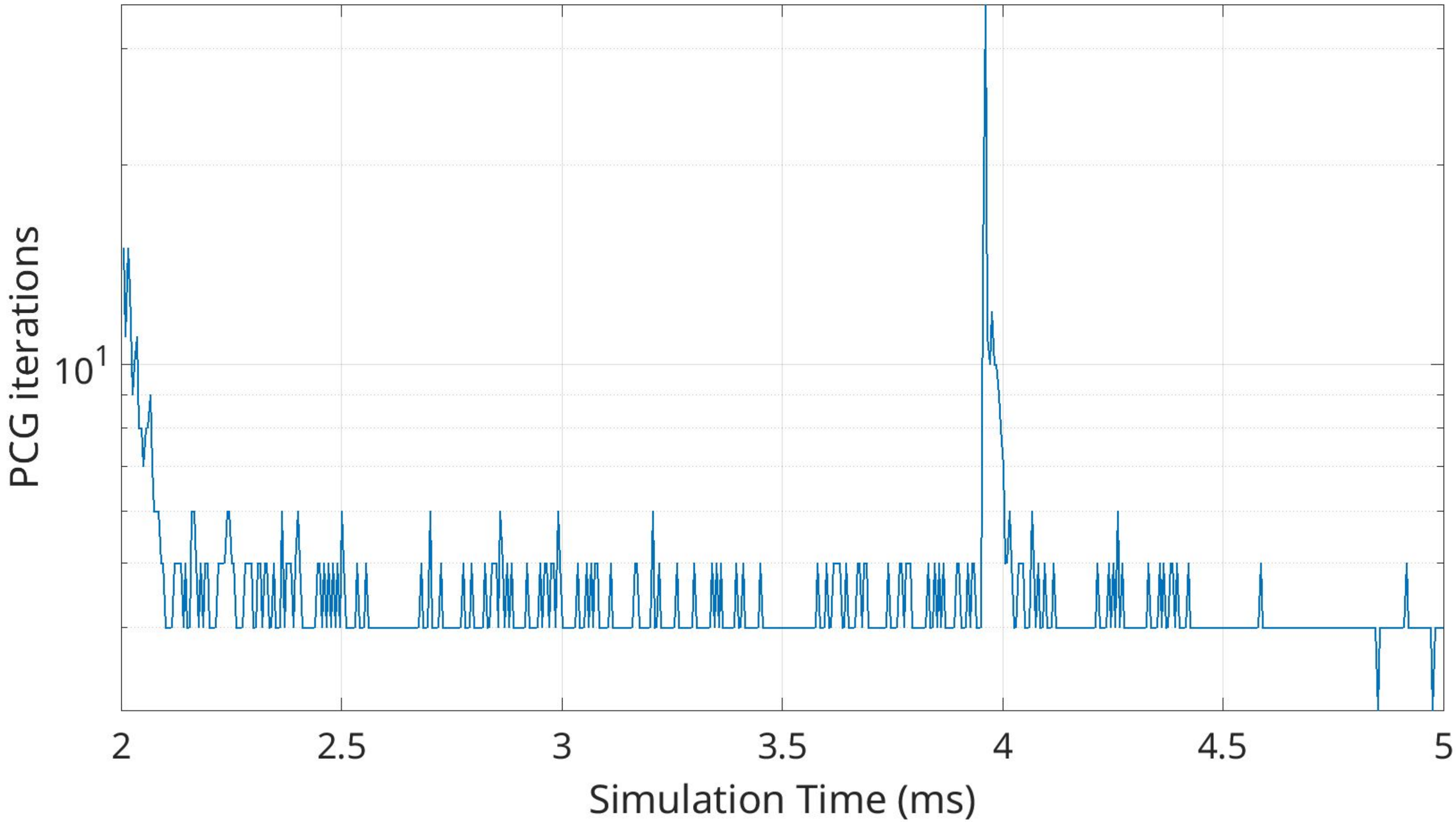
$\begin{bmatrix} W_i R_i r \\ 0 \end{bmatrix}$

Computed as described in [Dohrmann](Dohrmann)

## Interface in Ginkgo

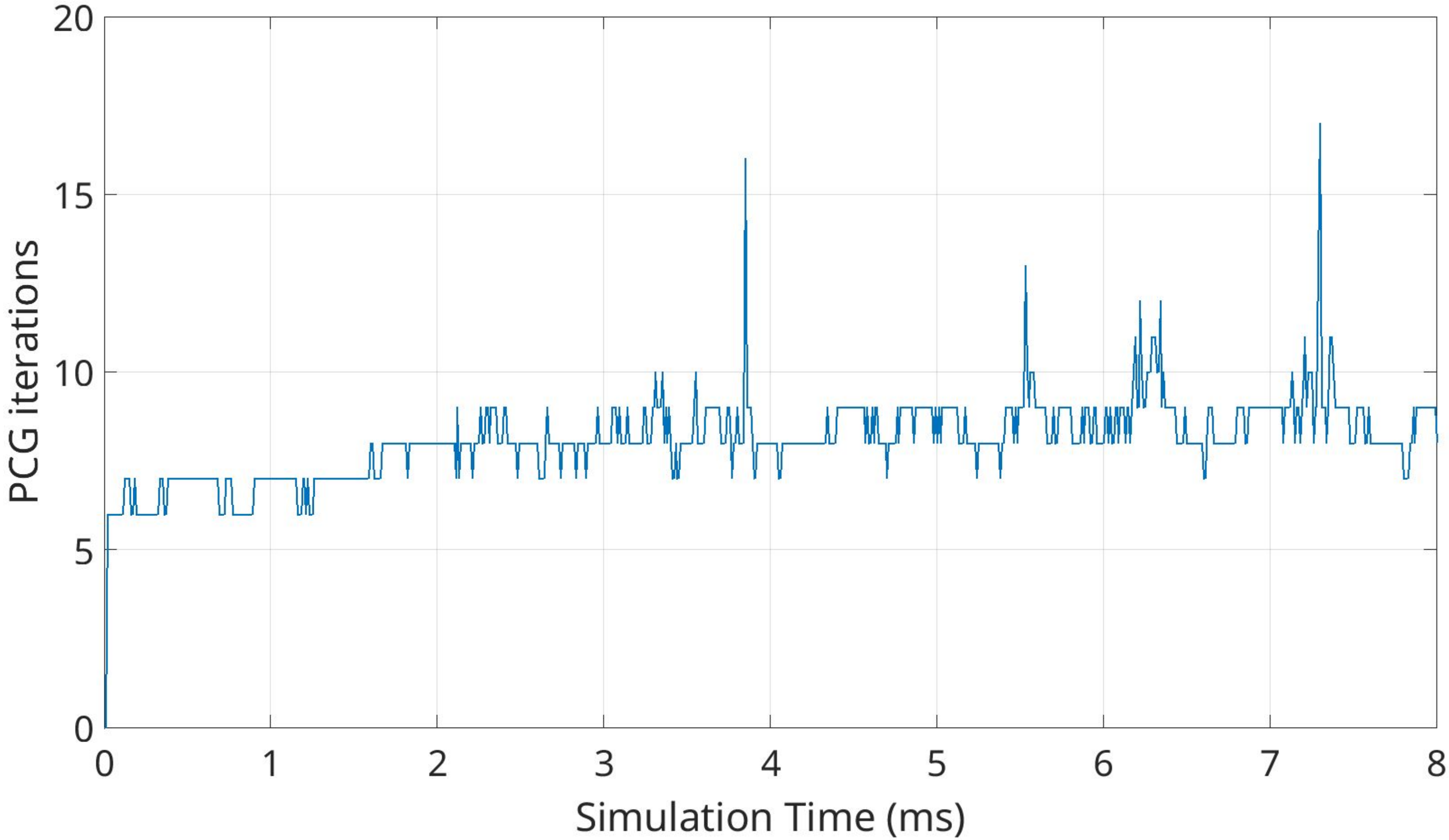
```
auto solver_factory = cg::build()
    .with_criteria(tol_stop, iter_stop)
    .with_preconditioner(bddc::build()
        .with_constraint_solver_factory(direct_factory)
        .with_local_solver_factory(direct_factory)
        .with_inner_solver_factory(direct_factory)
        .with_coarse_solver_factory(cg_factory)
        .with_static_condensation(true)
        .on(exec))
    .on(exec);
auto solver = solver_factory->generate(matrix);
```

# Convergence in Bidomain Simulations




8 ranks, noground bidom example in openCARP

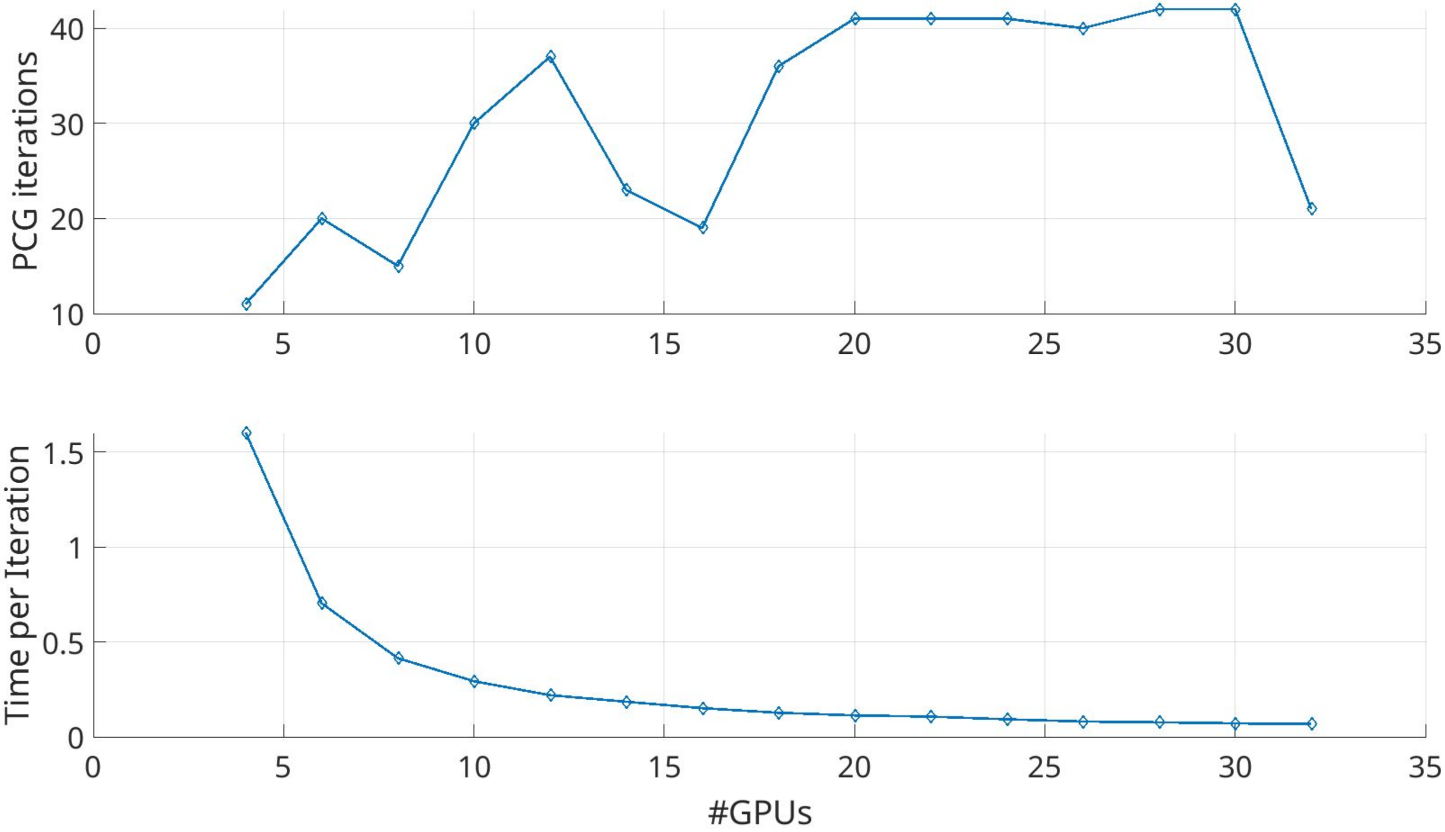
## Convergence in Bidomain Simulations



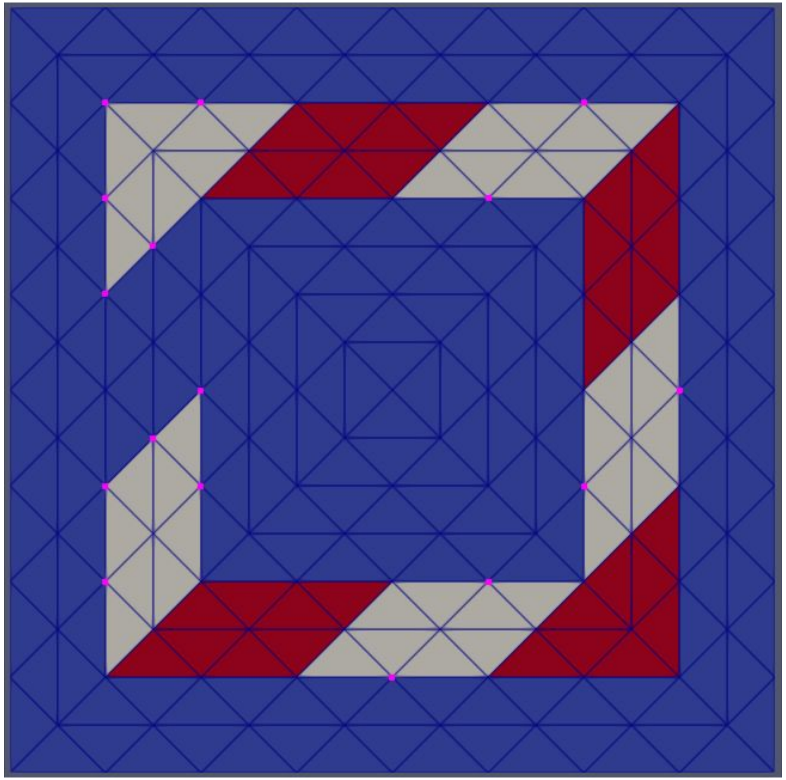

8 ranks, stimulation example in openCARP

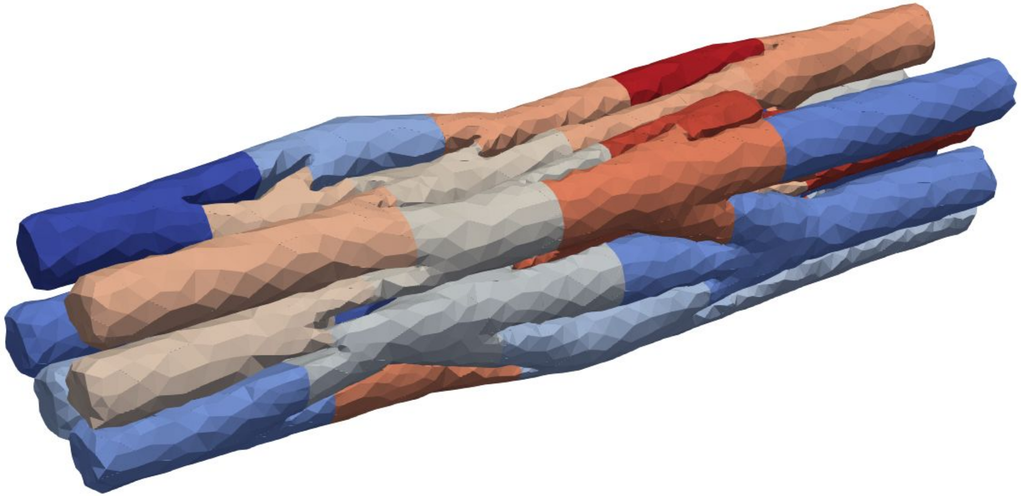
## Strong Scaling for Bidomain Matrix

- SPD stiffness matrix from noground_bidom example
- ~120k DOFs, ~1.4M nonzeros

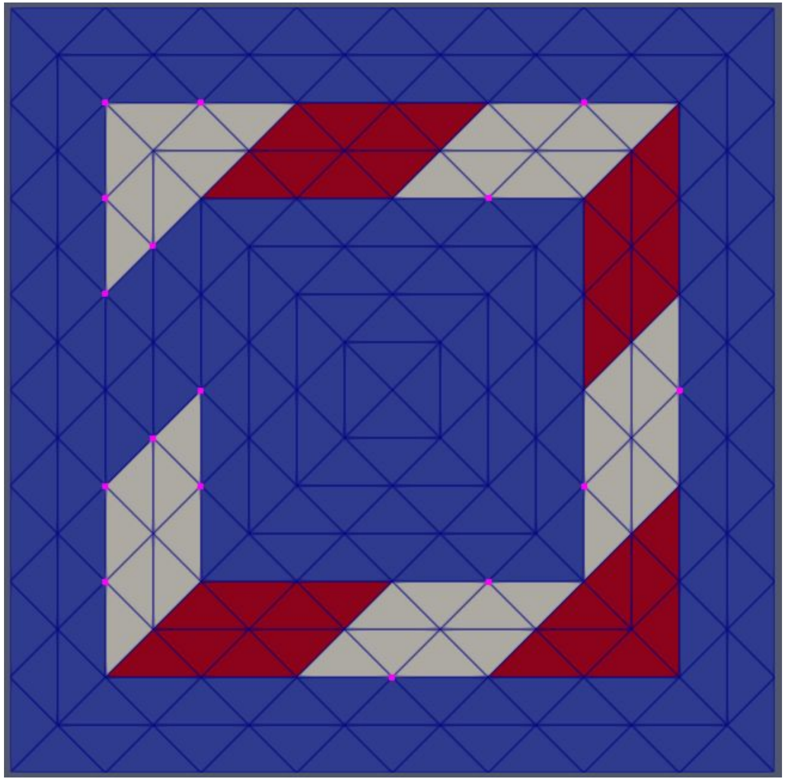


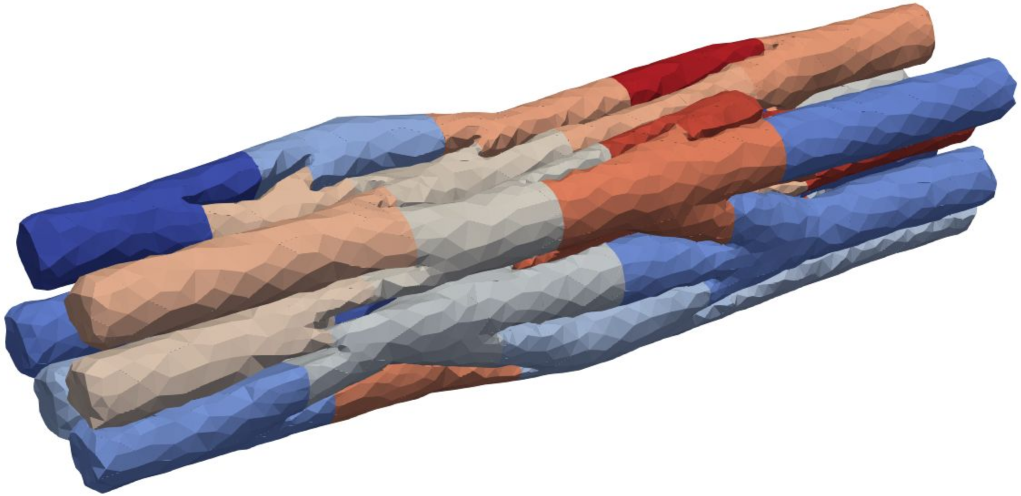
# EMI test cases - thank you Fatemeh for providing them!





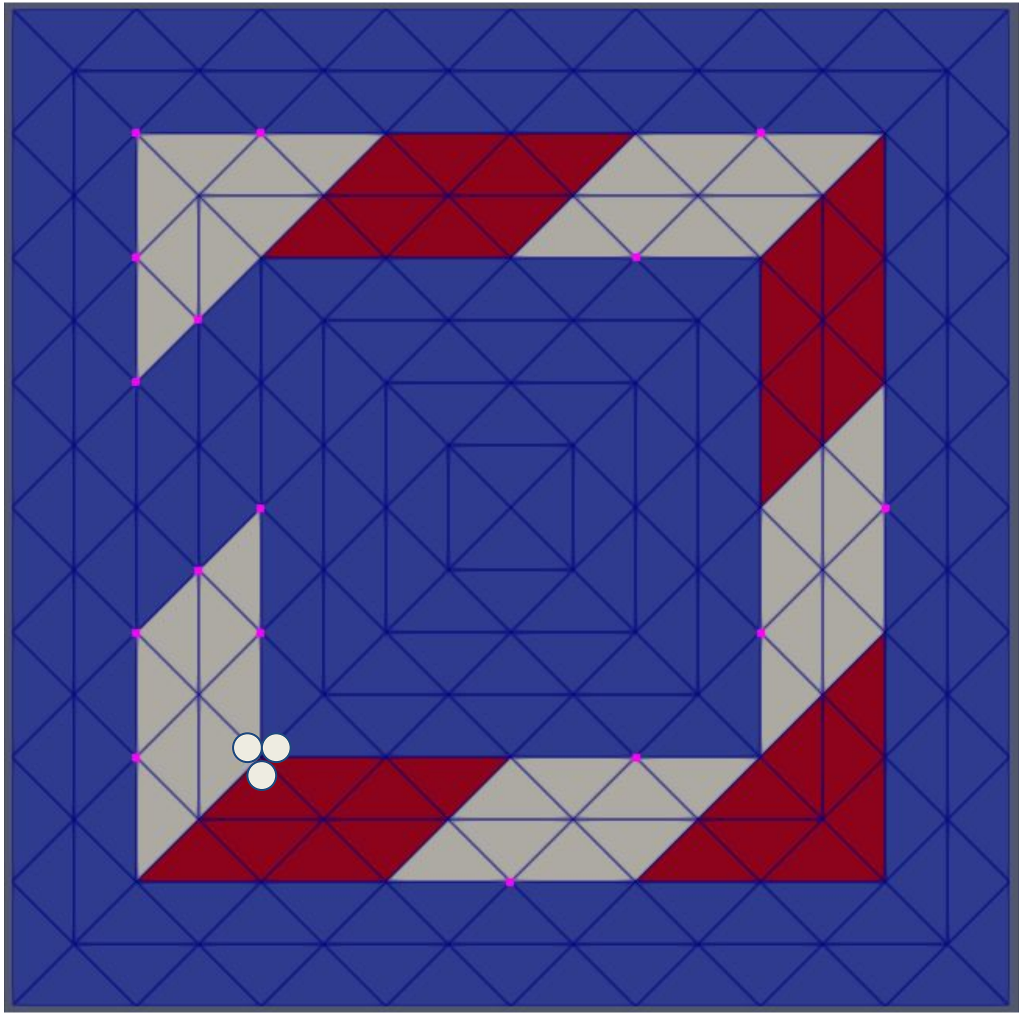
## EMI test cases - thank you Fatemeh for providing them!





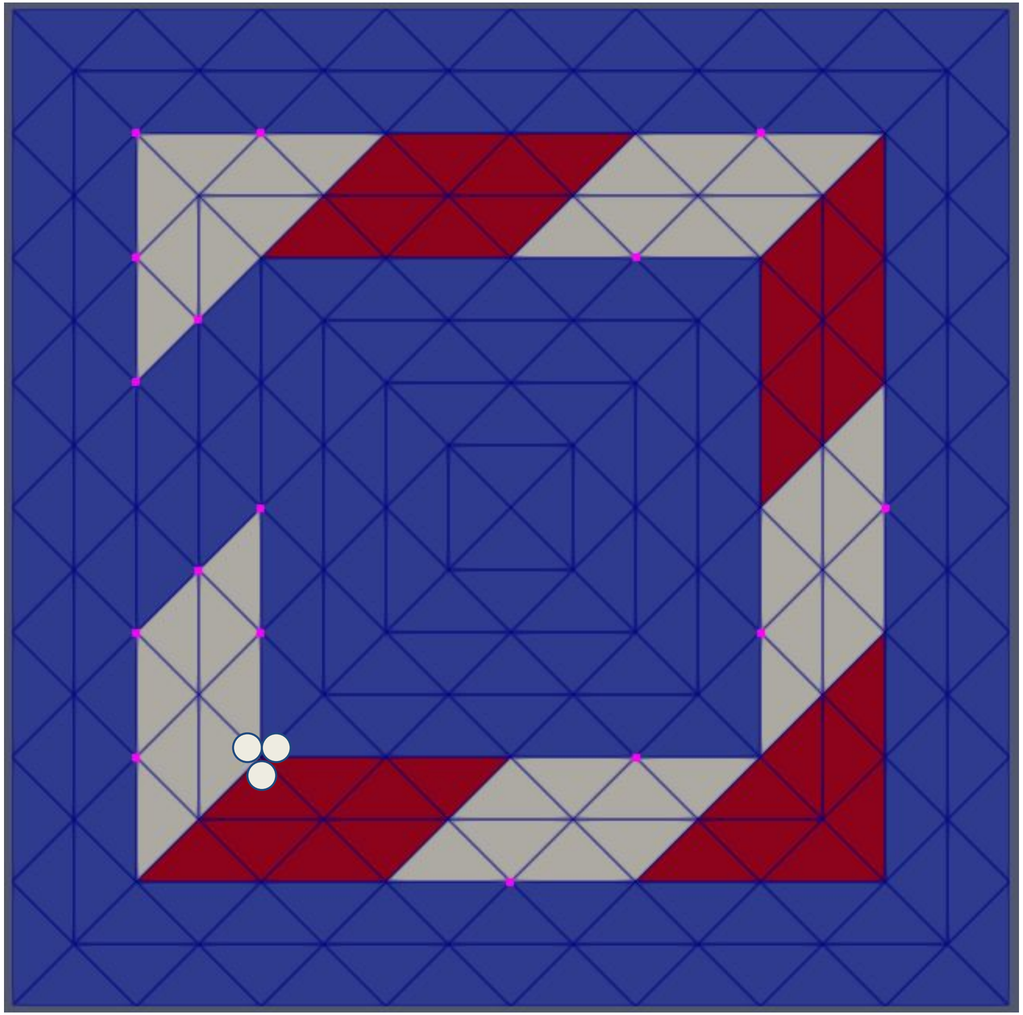
No convergence. This is disappointing!

## The coarse space is important!



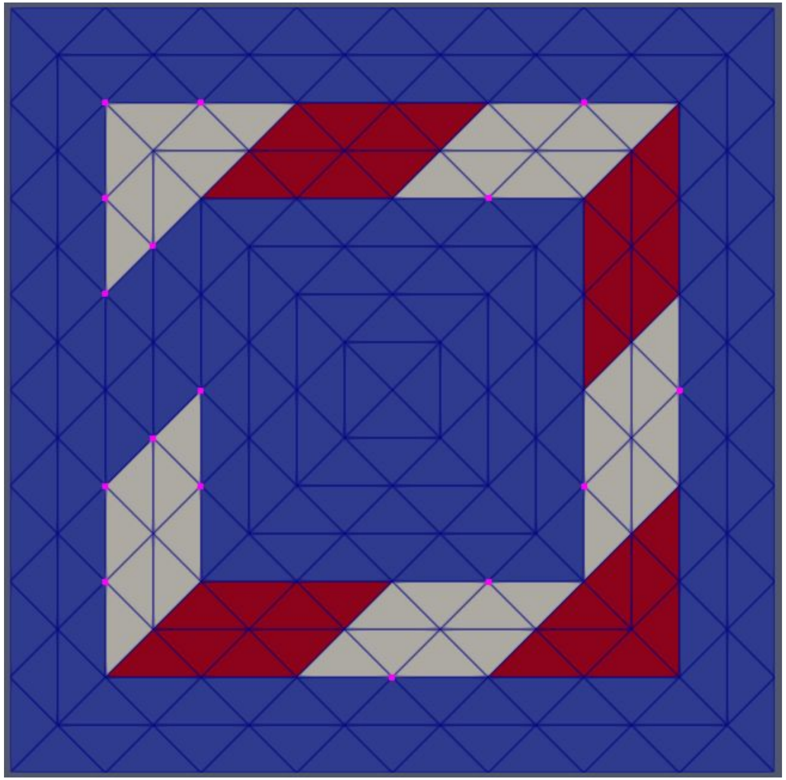
- Detection of
  - edges: dofs shared between exactly two subdomains
  - corners: dofs shared between more than two subdomains
- No notion between actual cells → only one interface between extra- / intracellular spaces.
- Splitting of dofs in the discretization makes corners no longer reliably detectable

# The coarse space is important!

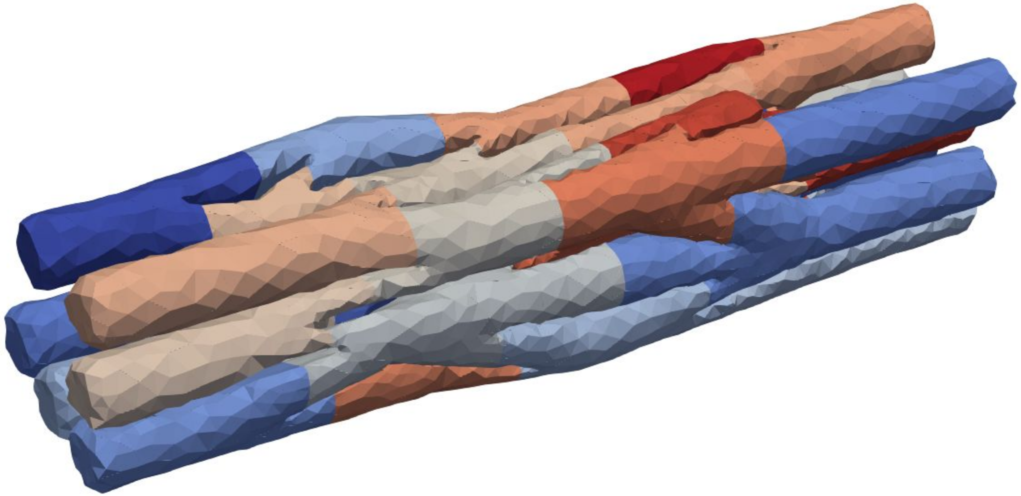


- Look at the matrix rows!
- interfaces / corners are strongly connected
- Detect an interface / corner:
  - Select starting node
  - add direct neighbours that are detected as interface nodes
  - grow until there are no new direct neighbours

## The coarse space is important!



22 CG iterations



147 CG iterations

# The coarse space is important!

```
std::vector<std::vector<int>> interface_dofs;
std::vector<std::vector<int>> ranks;

auto solver_factory = cg::build()
    .with_criteria(tol_stop, iter_stop)
    .with_preconditioner(bddc::build()
        .with_constraint_solver_factory(direct_factory)
        .with_local_solver_factory(direct_factory)
        .with_inner_solver_factory(direct_factory)
        .with_coarse_solver_factory(cg_factory)
        .with_static_condensation(true)
        .with_interface_dofs(interface_dofs)
        .with_interface_dof_ranks(ranks)
        .on(exec))
    .on(exec);
auto solver = solver_factory->generate(matrix);
```

## Mixed Precision - do we have to be precise everywhere?

$$M^{-1} = v_1 + v_2 + v_3$$

Coarse grid correction

Subdomain correction

Static condensation correction

$$v_1 = \sum_{i=1}^{N} R_i^T W_i \Phi_i R_{ci} A_c^{-1} r_c$$

$$r_c = R_{ci}^T \Phi_i^T W_i R_i r$$

$$v_2 = \sum_{i=1}^{N} R_i^T W_i z_i$$

$$\begin{bmatrix} A_i & C_i^T \\ C_i & 0 \end{bmatrix} \begin{bmatrix} z_i \\ \lambda_i \end{bmatrix} = \begin{bmatrix} W_i R_i r \\ 0 \end{bmatrix}$$

$$v_3 = \sum_{i=1}^{N} R_i^T R_{Ii}^T (R_{Ii} A_i R_{Ii}^T)^{-1} R_{Ii} R_i r_1$$

$$r_1 = r - A(v_1 + v_2)$$

Recent mixed precision research: keep residual computations in high precision, reduce precision elsewhere

# Mixed Precision - do we have to be precise everywhere?

$$M^{-1} = v_1 + v_2 + v_3$$

Coarse grid correction

$$v_1 = \sum_{i=1}^{N} R_i^T W_i \Phi_i R_{ci} A_c^{-1} r_c$$

$$r_c = R_{ci}^T \Phi_i^T W_i R_i r$$

Subdomain correction

$$v_2 = \sum_{i=1}^{N} R_i^T W_i z_i$$

$$\begin{bmatrix} A_i & C_i^T \\ C_i & 0 \end{bmatrix} \begin{bmatrix} z_i \\ \lambda_i \end{bmatrix} = \begin{bmatrix} W_i R_i r \\ 0 \end{bmatrix}$$

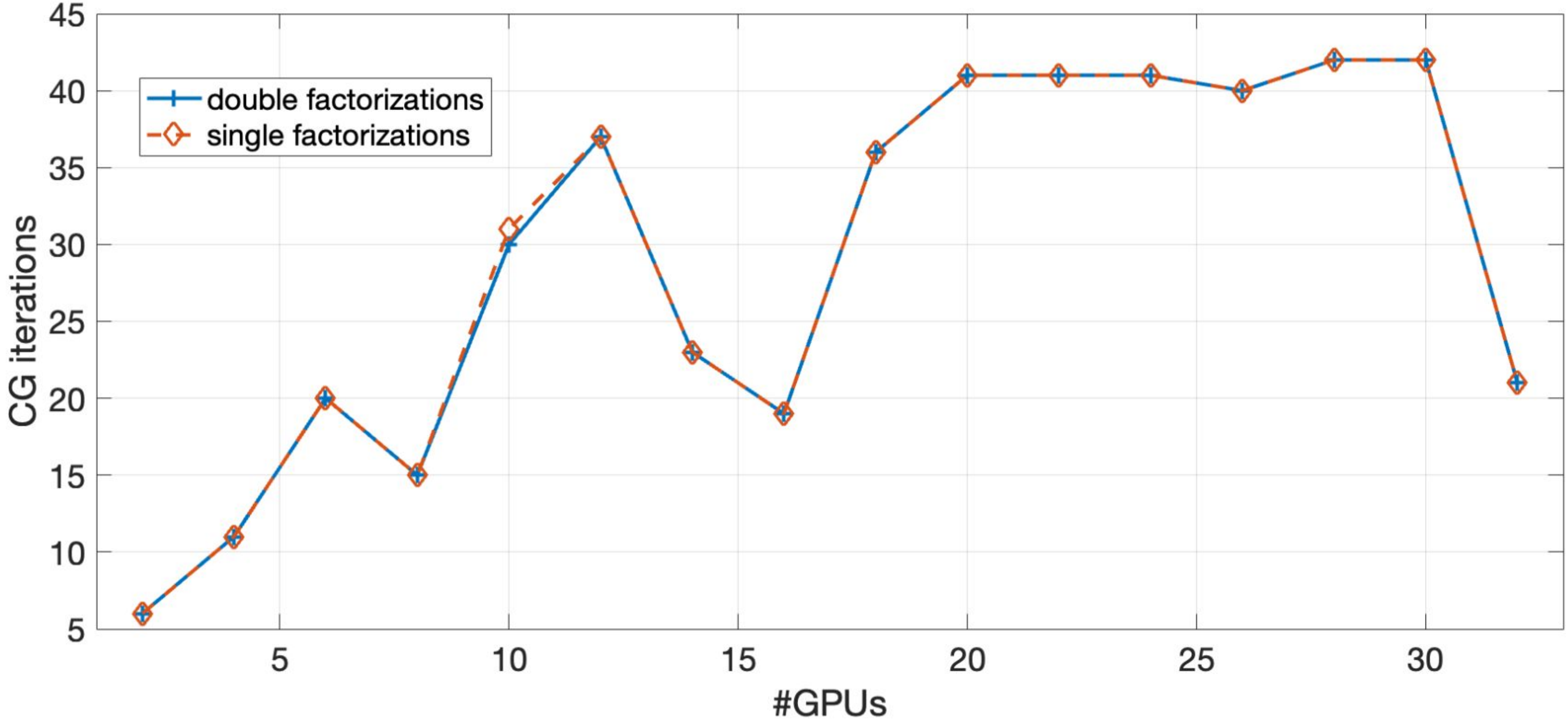
Static condensation correction

$$v_3 = \sum_{i=1}^{N} R_i^T R_{Ii}^T (R_{Ii} A_i R_{Ii}^T)^{-1} R_{Ii} R_i r_1$$

$$r_1 = r - A(v_1 + v_2)$$

Recent mixed precision research: keep residual computations in high precision, reduce precision elsewhere

## Mixed Precision - do we have to be precise everywhere?




- Reducing precision in local factorizations barely any impact on convergence
- Could save us some memory and (hopefully) time

## Mixed Precision - do we have to be precise everywhere?

```
auto solver_factory = cg::build()
    .with_criteria(tol_stop, iter_stop)
    .with_preconditioner(bddc::build()
        .with_constraint_solver_factory(direct_factory)
        .with_local_solver_factory(mixed_factory)
        .with_inner_solver_factory(mixed_factory)
        .with_coarse_solver_factory(cg_factory)
        .with_static_condensation(true)
        .on(exec))
    .on(exec);
auto solver = solver_factory->generate(matrix);
```

## Conclusion and Outlook

- Construct the coarse problem carefully!
- Promising first mixed precision results

What's next?

- Try mixed precision for EMI problems
- Implement coarse space construction in Ginkgo / get this information from openCARP?

Questions and or Suggestions?

## Thank you for your attention!

This project has received funding from the European High-Performance Computing Joint Undertaking EuroHPC (JU) under grant agreement No 955495. The JU receives support from the European Union's Horizon 2020 research and innovation programme and France, Italy, Germany, Austria, Norway, and Switzerland. The project was co-funded by the French National Research Agency ANR, the German Federal Ministry of Education and Research, the Italian ministry of economic development, the Swiss State Secretariat for Education, Research and Innovation, the Austrian Research Promotion Agency FFG, and the Research Council of Norway.