Fast and Robust Overlapping Schwarz Preconditioners in Trilinos

Highly Scalable Algorithms and Their Efficient Implementation

Alexander Heinlein

HPC Seminar Series, Center for Data and Simulation Science (CDS), Universität zu Köln, Köln, Germany, January 18, 2023

TU Delft

Outline

1. Exascale Computing & Trilinos

2. The FROSch (Fast and Robust Overlapping Schwarz) Package

3. Distributed Memory Parallelization – Scalability Results for FROSch Preconditioners

4. Node-Level Performance of FROSch Preconditioners

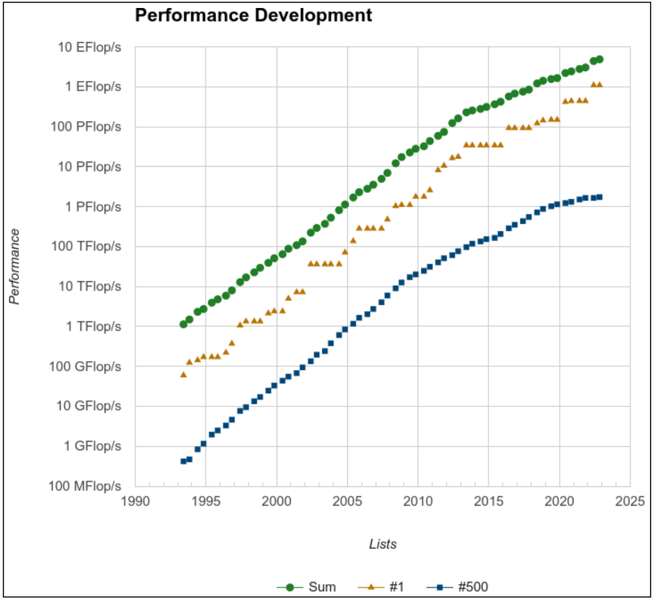
Exascale Computing & Trilinos

Exascale Computing

*For most scientific and engineering applications, Exascale implies 10^{18} IEEE 754 Double Precision (64-bit) operations (multiplications and/or additions) per second (**exaflops**). The **High Performance Linpack (HPL) benchmark**, which **solves a dense linear system using LU factorization with partial pivoting**, is the current benchmark by which the community measures the throughput of a computing system. **To be generally accepted as an Exascale system, a computer must exceed** 10^{18} **flops (1 exaflops) on the HPL benchmark**.*

Bergman, Keren, et al.
Exascale computing study: Technology challenges in achieving exascale systems.
Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep 15 (2008): 181.

TOP500 List (November 2022)



Taken from https://www.top500.org.

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** – HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, **AMD Instinct MI250X**, Slingshot-11 HPE DOE/SC/Oak Ridge National Laboratory, USA	**8,730,112**	**1,102.00**	1,685.65	21,100
2	**Supercomputer Fugaku** – Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science, Japan	**7,630,848**	442.01	537.21	29,899
3	**LUMI** – HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, **AMD Instinct MI250X**, Slingshot-11, HPE EuroHPC/CSC, Finland	**2,220,288**	309.10	428.70	6,016
4	**Leonardo** – BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, **NVIDIA A100 SXM4 64 GB**, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA, Italy	**1,463,616**	174.70	255.75	5,610
5	**Summit** – IBM Power System AC922, IBM POWER9 22C 3.07GHz, **NVIDIA Volta GV100**, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory, USA	**2,414,592**	148.60	200.79	10,096
...	...	...	...	...	...

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** – HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, **AMD Instinct MI250X**, Slingshot-11 HPE DOE/SC/Oak Ridge National Laboratory, USA	**8,730,112**	**1,102.00**	1,685.65	21,100
2	**Supercor** A64FX 4 RIKEN C			37.21	29,899
3	**LUMI** – 3rd Gene **MI250X**, Slingshot-11, HPE EuroHPC/CSC, Finland			28.70	6,016
4	**Leonardo** – BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, **NVIDIA A100 SXM4 64 GB**, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA, Italy	**1,463,616**	174.70	255.75	5,610
5	**Summit** – IBM Power System AC922, IBM POWER9 22C 3.07GHz, **NVIDIA Volta GV100**, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory, USA	**2,414,592**	148.60	200.79	10,096
...	...	...	...	...	...

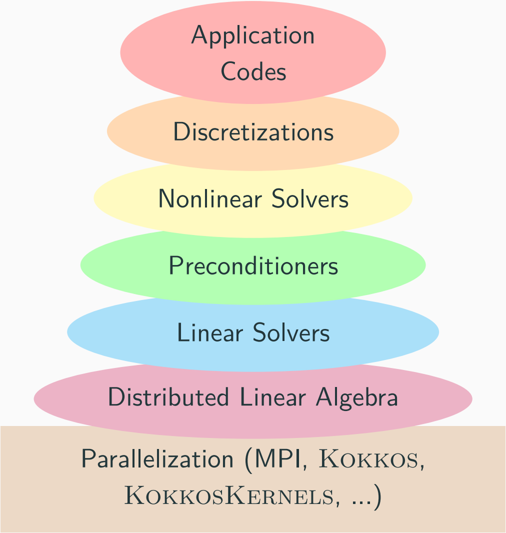
Out of a total of **8,730,112 computing cores**, **8,138,240 cores** are **GPU cores**.
→ **Almost all the performance in computing bandwidth is on the GPUs.**

What is Trilinos?

An Open-Source Library of Software for Scientific Computing

Mission statement (**Heroux et al. (2005)**): *"The* TRILINOS *Project is an effort to facilitate the design, development, integration, and ongoing support of mathematical software libraries and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems on new and emerging high-performance computing (HPC) architectures".*

Layers of a Trilinos-Based Application



Application Codes

Discretizations

Nonlinear Solvers

Preconditioners

Linear Solvers

Distributed Linear Algebra

Parallelization (MPI, Kokkos, KokkosKernels, ...)

Why Using Trilinos?

Wide range of functionality

Data services	Vectors, matrices, graphs and similar data containers, and related operations
Linear and eigenproblem solvers	For large, distributed systems of equations
Nonlinear solvers and analysis tools	Includes basic nonlinear approaches, continuation methods and similar
Discretizations	Tools for the discretization of integral and differential equations
Framework	Tools for building, testing, and integrating TRILINOS capabilities

Portable parallelism

TRILINOS is targeted for all major parallel architectures, including

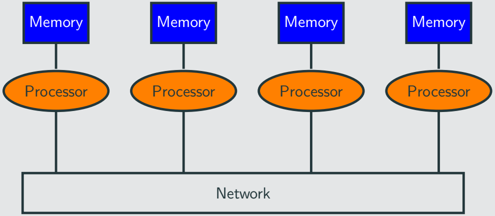
- distributed-memory using the Message Passing Interface (MPI),
- multicore using a variety of common approaches,
- accelerators using common and emerging approaches, and
- vectorization.

" . . . as long as a given algorithm and problem size contain enough latent parallelism, ***the same Trilinos source code*** *can be compiled and execution on* ***any reasonable combination of distributed, multicore, accelerator and vectorizing computing devices****."* — Trilinos Website

Parallelization in Trilinos

Distributed-memory parallelization

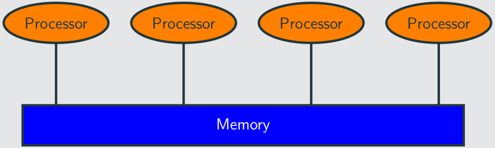
- **Process**-based parallelization
- **Each processor** has its **own internal memory**
- ⊕ No memory access conflicts
- ⊖ Requires *(possibly slow)* data exchange through a network



MPI

Shared-memory parallelization

- **Thread**-based parallelization
- **All processors** access a **shared memory**
- ⊕ Changes in shared memory are visible to all
- ⊖ Memory access conflicts



CPU	GPU
OpenMP **Pthreads**	**CUDA**

Parallelization and Performance Portability in Trilinos

Distributed-memory parallelization (MPI)

MPI parallelization is provided through the *parallel linear algebra framework*:

- At the moment, there are two different linear algebra frameworks/packages, the older **Epetra** package and the more recent **Tpetra** package.
- The linear algebra frameworks both provide parallel implementations of
  - vectors,
  - sparse matrices,
  - redistributors,
  - and more. . .
- Based on Epetra and Tpetra, Trilinos currently provides two stacks of packages, providing a similar range of functionality.
- Tpetra is built upon Kokkos; see right.

Shared-memory parallelization (X)

A systematic framework for **shared-memory parallelization** is provided by the Kokkos programming model:

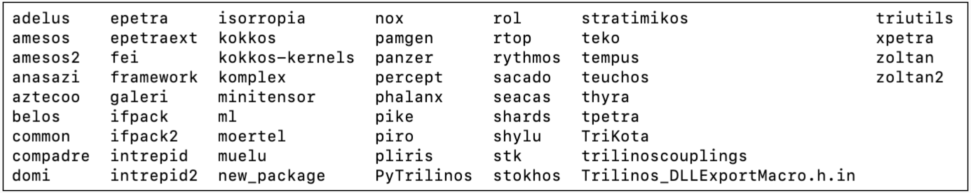
- **Kokkos** implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms.
- **KokkosKernels** implements local computational kernels for linear algebra and graph operations, using the Kokkos programming model.
- Support for CUDA, HPX, OpenMP and Pthreads.
- Tpetra automatically provides access to the functionality of Kokkos.

Overview of Trilinos Packages

Trilinos is a collection of more than 50 software packages:

- Each Trilinos package is a *self-contained, independent piece of software* with its *own set of requirements, its own development team*[1] *and group of users*.
- However, there are often certain *dependencies between different* Trilinos *packages*. Some Trilinos packages also *depend on third party libraries (TPLs)*.
- Generally, a *certain degree of interoperability* of the different Trilinos packages is provided.

Contents of `trilinos/packages`:



```
adelus    epetra     isorropia      nox        rol      stratimikos                   triutils
amesos    epetraext  kokkos         pamgen     rtop     teko                          xpetra
amesos2   fei        kokkos-kernels panzer     rythmos  tempus                        zoltan
anasazi   framework  komplex        percept    sacado   teuchos                       zoltan2
aztecoo   galeri     minitensor     phalanx    seacas   thyra
belos     ifpack     ml             pike       shards   tpetra
common    ifpack2    moertel        piro       shylu    TriKota
compadre  intrepid   muelu          pliris     stk      trilinoscouplings
domi      intrepid2  new_package    PyTrilinos stokhos  Trilinos_DLLExportMacro.h.in
```

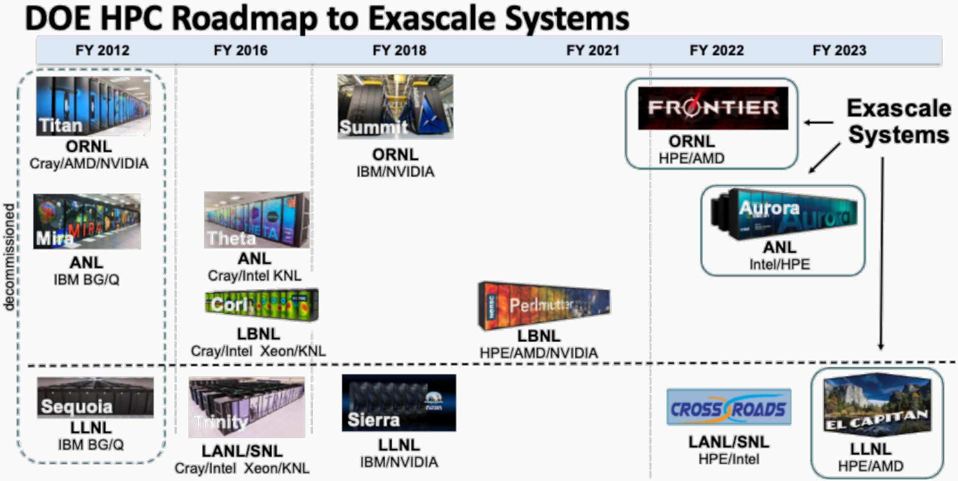
Trilinos Packages

	MPI (Epetra-based)	MPI+X (Tpetra-based)
Linear algebra	**Epetra** & **EpetraExt**	**Tpetra**
Direct sparse solvers	**Amesos**	**Amesos2**
Iterative solvers	**AztecOO**	**Belos**
Preconditioners:		
• One-level (incomplete) factorization	**Ifpack**	**Ifpack2**
• Multigrid	**ML**	**MueLu**
• Domain decomposition		**ShyLU**
Eigenproblem solvers		**Anasazi**
Nonlinear solvers	**NOX** & **LOCA**	
Partitioning	**Isorropia** & **Zoltan**	**Zoltan2**
Example problems	**Galeri**	
Performance portability		**Kokkos** & **KokkosKernels**
Interoperability	**Stratimikos** & **Thyra**	
Tools	**Teuchos**	
⋮	⋮	⋮

- Packages, that do not depend on Epetra or Tpetra work in both software stacks, e.g., Galeri, NOX & LOCA, Teuchos
- More details on https://trilinos.github.io.

Trilinos & Exascale Computing

The development of TRILINOS towards Exascale computing is strongly influenced by **U.S. Exascale systems** as well as the **Exascale Computing Project**.



See http://e4s.io

Trilinos & Exascale Computing

The development of Trilinos towards Exascale computing is strongly influenced by **U.S. Exascale systems** as well as the **Exascale Computing Project**.

Exascale Computing Project

The Exascale Computing Project (ECP) is a collaborative effort of two US Department of Energy (DOE) organizations – the Office of Science (DOE-SC) and the National Nuclear Security Administration (NNSA).

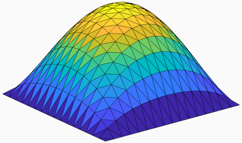
https://www.exascaleproject.org/

The ECP is commissioned to provide new scientific software capabilities on the frontier of algorithms, software and hardware

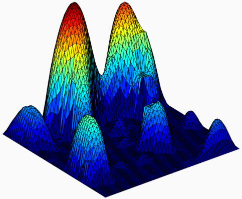
- ECP uses platforms to foster collaboration and cooperation as we head into the frontier
- ECP has two primary software platforms:
  - **E4S** (Extreme-scale Scientific Software Stack): a comprehensive portfolio of ECP-sponsored products and dependencies
  - **SDKs** (Software Development Kits): Domain-specific collaborative and aggregate product development of similar capabilities

The FROSch (Fast and Robust Overlapping Schwarz) Package

Solving A Model Problem



$\alpha(x) = 1$



heterogeneous $\alpha(x)$

Consider a **diffusion model problem**:

$$-\nabla \cdot (\alpha(x) \nabla u(x)) = f \quad \text{in } \Omega = [0,1]^2,$$
$$u = 0 \quad \text{on } \partial\Omega.$$

Discretization using finite elements yields a **sparse** linear system of equations

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f}.$$

Direct solvers

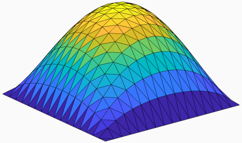
For fine meshes, solving the system using a direct solver is not feasible due to **superlinear complexity and memory cost**.

Iterative solvers

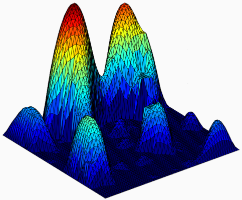
Iterative solvers are efficient for solving sparse linear systems of equations, however, the **convergence rate generally depends on the condition number** $\kappa(\boldsymbol{A})$. It deteriorates, e.g., for

- fine meshes, that is, small element sizes h
- large contrasts $\frac{\max_x \alpha(x)}{\min_x \alpha(x)}$

Solving A Model Problem



$\alpha(x) = 1$



heterogeneous $\alpha(x)$

Consider a **diffusion model problem**:

$$-\nabla \cdot (\alpha(x)\nabla u(x)) = f \quad \text{in } \Omega = [0,1]^2,$$
$$u = 0 \quad \text{on } \partial\Omega.$$

Discretization using finite elements yields a **sparse** linear system of equations

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f}.$$

Direct solvers

For fine meshes, solving the system using a direct solver is not feasible due to **superlinear complexity and memory cost**.

Iterative solvers

Iterative solvers are efficient for solving sparse linear systems of equations, however, the **convergence rate generally depends on the condition number** $\kappa(\boldsymbol{A})$. It deteriorates, e.g., for

- fine meshes, that is, small element sizes h
- large contrasts $\frac{\max_x \alpha(x)}{\min_x \alpha(x)}$

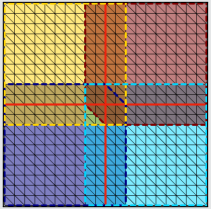
$\Rightarrow$ We introduce a preconditioner $\boldsymbol{M}^{-1} \approx \boldsymbol{A}^{-1}$ to improve the condition number:

$$\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{u} = \boldsymbol{M}^{-1}\boldsymbol{f}$$

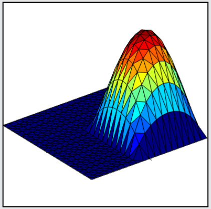
Two-Level Schwarz Preconditioners

One-level Schwarz preconditioner

Overlap $\delta = 1h$



Solution of local problem



Based on an **overlapping domain decomposition**, we define a **one-level Schwarz operator**

$$M_{\text{OS-1}}^{-1} K = \sum\nolimits_{i=1}^{N} R_i^T K_i^{-1} R_i K,$$

where R_i and R_i^T are restriction and prolongation operators corresponding to Ω_i', and $K_i := R_i K R_i^T$.

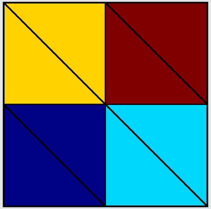
Condition number estimate:

$$\kappa\left(M_{\text{OS-1}}^{-1} K\right) \leq C \left(1 + \frac{1}{H\delta}\right)$$

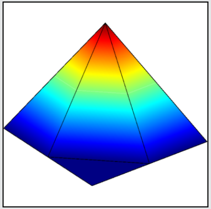
with subdomain size H and overlap width δ.

Lagrangian coarse space

Coarse triangulation



Coarse solution



The **two-level overlapping Schwarz operator** reads

$$M_{\text{OS-2}}^{-1} K = \underbrace{\Phi K_0^{-1} \Phi^T K}_{\text{coarse level -- global}} + \underbrace{\sum\nolimits_{i=1}^{N} R_i^T K_i^{-1} R_i K}_{\text{first level -- local}},$$

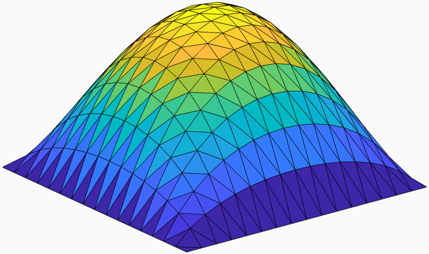
where Φ contains the coarse basis functions and $K_0 := \Phi^T K \Phi$; cf., e.g., Toselli, Widlund (2005). The construction of a Lagrangian coarse basis requires a coarse triangulation.

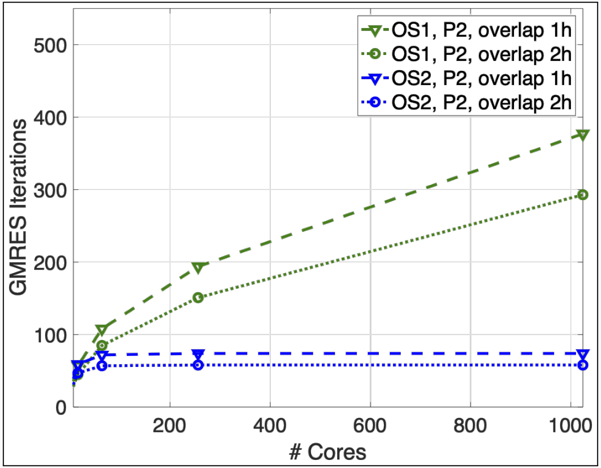
Condition number estimate:

$$\kappa\left(M_{\text{OS-2}}^{-1} K\right) \leq C \left(1 + \frac{H}{\delta}\right)$$

One- Vs Two-Level Schwarz Preconditioners

Diffusion model problem in two dimensions, # subdomains = # cores, $H/h = 100$





FROSch (Fast and Robust Overlapping Schwarz) Framework in Trilinos

Software

- Object-oriented C++ domain decomposition solver framework with MPI-based distributed memory parallelization
- Part of TRILINOS with support for both parallel linear algebra packages EPETRA and TPETRA
- Node-level parallelization and performance portability on CPU and GPU architectures through KOKKOS
- Accessible through unified TRILINOS solver interface STRATIMIKOS

Methodology

- Parallel scalable multi-level Schwarz domain decomposition preconditioners
- Algebraic construction based on the parallel distributed system matrix
- Extension-based coarse spaces

Team (active)

- Alexander Heinlein (TU Delft)
- Siva Rajamanickam (Sandia)
- Friederike Röver (TUBAF)
- Axel Klawonn (Uni Cologne)
- Oliver Rheinbach (TUBAF)
- Ichitaro Yamazaki (Sandia)

FROSch (Fast and Robust Overlapping Schwarz) Framework in Trilinos

Software

- Object-oriented C++ domain decomposition solver framework with MPI-based distributed memory parallelization
- Part of Trilinos with support for both parallel linear algebra packages Epetra and Tpetra
- Node-level parallelization and performance portability on CPU and GPU architectures through Kokkos
- Accessible through unified Trilinos solver interface Stratimikos

Methodology

- **Parallel scalable** multi-level Schwarz domain decomposition preconditioners
- **Algebraic construction** based on the parallel distributed system matrix
- Extension-based coarse spaces

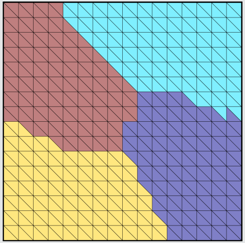
Team (active)

- Alexander Heinlein (TU Delft)
- Siva Rajamanickam (Sandia)
- Friederike Röver (TUBAF)
- Axel Klawonn (Uni Cologne)
- Oliver Rheinbach (TUBAF)
- Ichitaro Yamazaki (Sandia)

Algorithmic Framework for FROSch Overlapping Domain Decompositions

Overlapping domain decomposition

In FROSch, the overlapping subdomains $\Omega_1', \ldots, \Omega_N'$ are constructed by **recursively adding layers of elements** to the nonoverlapping subdomains; this can be performed based on the sparsity pattern of K.

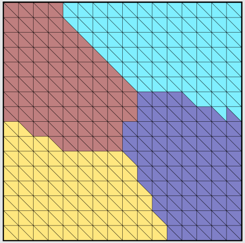


Nonoverlapping DD

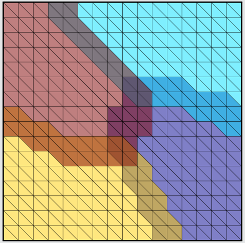
Algorithmic Framework for FROSch Overlapping Domain Decompositions

Overlapping domain decomposition

In FROSCH, the overlapping subdomains $\Omega'_1, \ldots, \Omega'_N$ are constructed by **recursively adding layers of elements** to the nonoverlapping subdomains; this can be performed based on the sparsity pattern of K.



Nonoverlapping DD

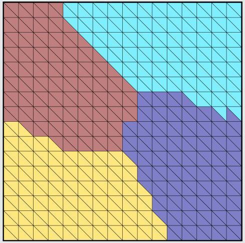


Overlap $\delta = 1h$

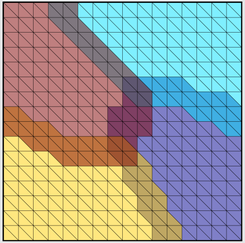
Algorithmic Framework for FROSch Overlapping Domain Decompositions

Overlapping domain decomposition

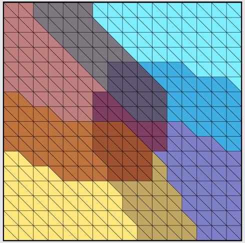
In FROSch, the overlapping subdomains $\Omega_1', \ldots, \Omega_N'$ are constructed by **recursively adding layers of elements** to the nonoverlapping subdomains; this can be performed based on the sparsity pattern of K.



Nonoverlapping DD



Overlap $\delta = 1h$

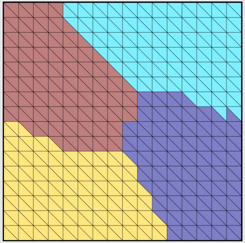


Overlap $\delta = 2h$

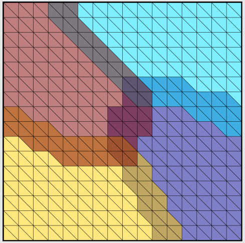
Algorithmic Framework for FROSch Overlapping Domain Decompositions

Overlapping domain decomposition

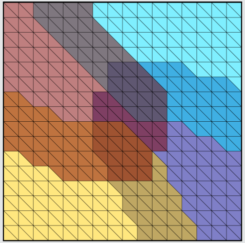
In FROSch, the overlapping subdomains $\Omega_1', \ldots, \Omega_N'$ are constructed by **recursively adding layers of elements** to the nonoverlapping subdomains; this can be performed based on the sparsity pattern of K.



Nonoverlapping DD



Overlap $\delta = 1h$



Overlap $\delta = 2h$

Computation of the overlapping matrices

The overlapping matrices

$$\boldsymbol{K}_i = \boldsymbol{R}_i \boldsymbol{K} \boldsymbol{R}_i^T$$

can easily be extracted from K since R_i is just a **global-to-local index mapping**.

Algorithmic Framework for FROSch Coarse Spaces

FROSch preconditioners use **algebraic coarse spaces** that are constructed in **four algorithmic steps**:

1. Identification of the **domain decomposition interface**
2. Construction of a **partition of unity (POU)** on the interface
3. Computation of a **coarse basis on the interface**
4. Harmonic extensions into the interior to obtain a **coarse basis** on the whole domain

Algorithmic Framework for FROSch Coarse Spaces

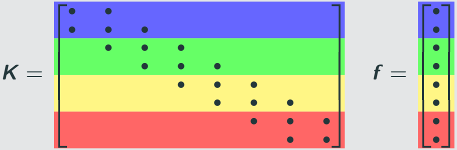
FROSch preconditioners use **algebraic coarse spaces** that are constructed in **four algorithmic steps**:

1. Identification of the **domain decomposition interface**
2. Construction of a **partition of unity (POU)** on the interface
3. Computation of a **coarse basis on the interface**
4. Harmonic extensions into the interior to obtain a **coarse basis** on the whole domain

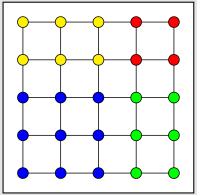
Identification of the domain decomposition interface

If not provided by the user, FROSch will construct a **repeated map** where the interface (Γ) nodes are shared between processes from the parallel distribution of the matrix rows (**distributed map**).

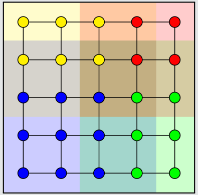
Then, FROSch automatically identifies vertices, edges, and (in 3D) faces, by the multiplicities of the nodes.



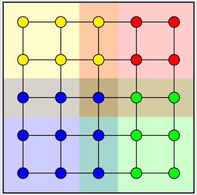
distributed map



overlapping map



repeated map



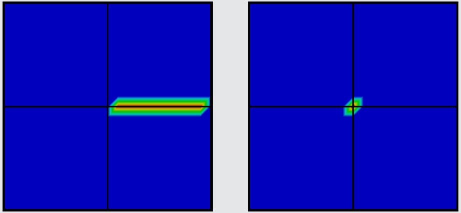
Algorithmic Framework for FROSch Coarse Spaces

FROSch preconditioners use **algebraic coarse spaces** that are constructed in **four algorithmic steps**:

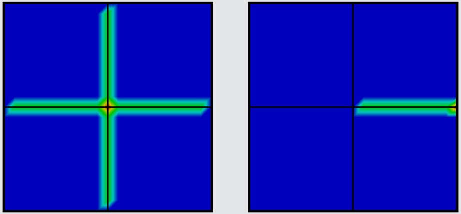
1. Identification of the **domain decomposition interface**
2. Construction of a **partition of unity (POU)** on the interface
3. Computation of a **coarse basis on the interface**
4. Harmonic extensions into the interior to obtain a **coarse basis** on the whole domain

Construction of a partition of unity on the interface

vertices, edges, and (in 3D) faces



overlapping vertex components

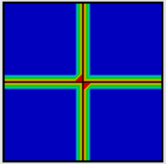


We construct a **partition of unity (POU)** $\{\pi_i\}_i$ with

$$\sum_i \pi_i = 1$$

on the interface Γ.

⇒

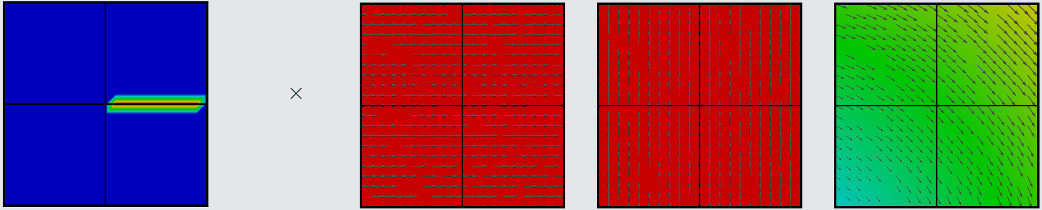


Algorithmic Framework for FROSch Coarse Spaces

FROSch preconditioners use **algebraic coarse spaces** that are constructed in **four algorithmic steps**:

1. Identification of the **domain decomposition interface**
2. Construction of a **partition of unity (POU)** on the interface
3. Computation of a **coarse basis on the interface**
4. Harmonic extensions into the interior to obtain a **coarse basis** on the whole domain

Computation of a coarse basis on the interface



interface POU function

null space basis (linear elasticity: **translations**, **linearized rotation(s)**)

For each partition of unity function π_i, we compute a basis for the space

$$\text{span}\left(\{\pi_i \times z_j\}_j\right),$$

where $\{z_j\}_j$ is a null space basis. In case of **linear dependencies**, we perform a **local QR factorization** to construct a basis.

This yields an **interface coarse basis** Φ_Γ.

The linearized rotation

$$\begin{bmatrix} y \\ -x \end{bmatrix}$$

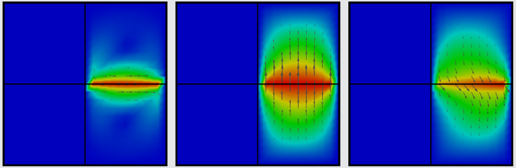
depends on coordinates (geometric information).

Algorithmic Framework for FROSch Coarse Spaces

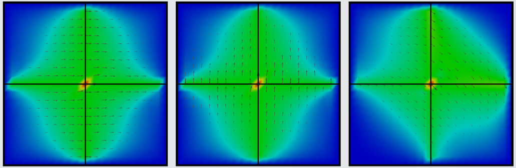
FROSch preconditioners use algebraic coarse spaces that are constructed in **four algorithmic steps**:

1. Identification of the **domain decomposition interface**
2. Construction of a **partition of unity (POU)** on the interface
3. Computation of a **coarse basis on the interface**
4. Harmonic extensions into the interior to obtain a **coarse basis** on the whole domain

Harmonic extensions into the interior



edge coarse basis functions



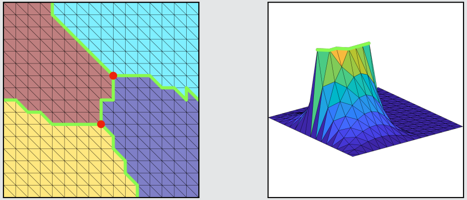
vertex component basis functions

For each **interface coarse basis function**, we compute the interior values Φ_I by computing **harmonic / energy-minimizing extensions**:

$$\Phi = \begin{bmatrix} -\boldsymbol{K}_{II}^{-1} \boldsymbol{K}_{\Gamma I}^{T} \Phi_\Gamma \\ \Phi_\Gamma \end{bmatrix} = \begin{bmatrix} \Phi_I \\ \Phi_\Gamma \end{bmatrix}.$$

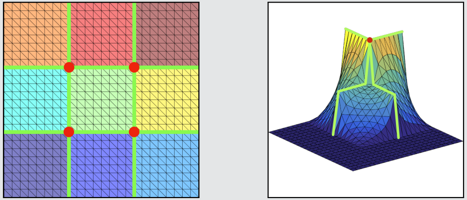
Examples of Extension-Based Coarse Spaces

GDSW (Generalized Dryja–Smith–Widlund)



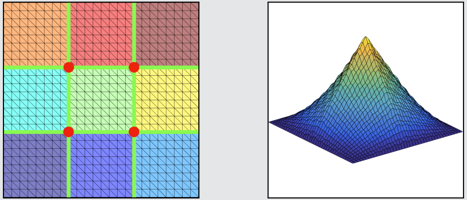
- Dohrmann, Klawonn, Widlund (2008)
- Dohrmann, Widlund (2009, 2010, 2012)

RGDSW (Reduced dimension GDSW)



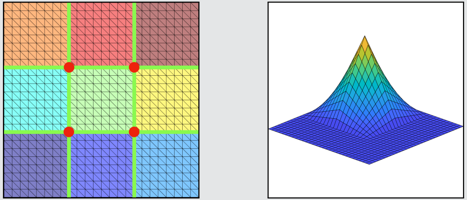
- Dohrmann, Widlund (2017)
- H., Klawonn, Knepper, Rheinbach, Widlund (2022)

MsFEM (Multiscale Finite Element Method)



- Hou (1997), Efendiev and Hou (2009)
- Buck, Iliev, and Andrä (2013)
- H., Klawonn, Knepper, Rheinbach (2018)

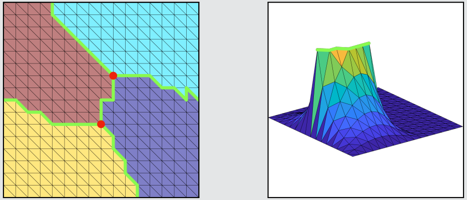
Q1 Lagrangian / piecewise bilinear



Piecewise linear interface partition of unity functions and a **structured domain decomposition**.

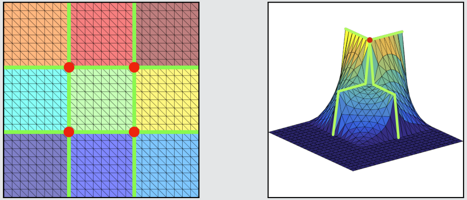
Examples of Extension-Based Coarse Spaces

GDSW (Generalized Dryja–Smith–Widlund)



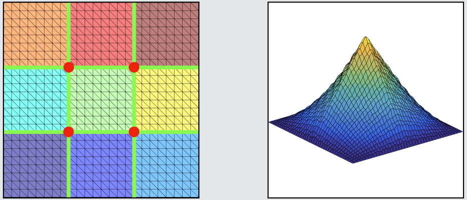
- Dohrmann, Klawonn, Widlund (2008)
- Dohrmann, Widlund (2009, 2010, 2012)

RGDSW (Reduced dimension GDSW)



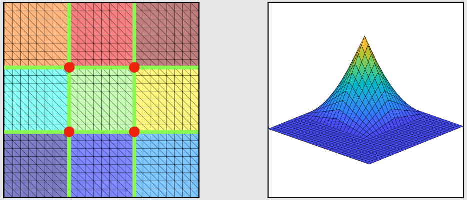
- Dohrmann, Widlund (2017)
- H., Klawonn, Knepper, Rheinbach, Widlund (2022)

MsFEM (Multiscale Finite Element Method)



- Hou (1997), Efendiev and Hou (2009)
- Buck, Iliev, and Andrä (2013)
- H., Klawonn, Knepper, Rheinbach (2018)

Q1 Lagrangian / piecewise bilinear

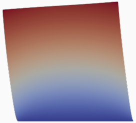


Piecewise linear interface partition of unity functions and a **structured domain decomposition**.

Distributed Memory Parallelization – Scalability Results for FROSch Preconditioners

Algebraic FROSch Preconditioners for Elasticity

$$\begin{aligned} \operatorname{div} \boldsymbol{\sigma} &= (0,-100,0)^T && \text{in } \Omega := [0,1]^3, \\ \boldsymbol{u} &= 0 && \text{on } \partial\Omega_D := \{0\} \times [0,1]^2, \\ \boldsymbol{\sigma} \cdot \boldsymbol{n} &= 0 && \text{on } \partial\Omega_N := \partial\Omega \setminus \partial\Omega_D \end{aligned}$$



St. Venant Kirchhoff material, P2 finite elements, $H/h = 9$; implementation in `FEDDLib`. (timings: setup + solve = **total**)

prec.	type	#cores	64	512	4 096
GDSW	rotations	#its.	**16.3**	**17.3**	**19.3**
		time	40.1 + 5.9 = **46.0**	55.0 + 8.5 = **63.5**	223.3 + 24.4 = **247.7**
	no rotations	#its.	**24.5**	**29.3**	**32.3**
		time	32.5 + 8.4 = **40.9**	38.4 + 11.8 = **46.7**	102.2 + 20.0 = **122.2**
	fully algebraic	#its.	**57.5**	**74.8**	**78.0**
		time	42.0 + 20.5 = **62.5**	46.0 + 29.9 = **75.9**	124.8 + 50.5 = **175.3**
RGDSW	rotations	#its.	**18.8**	**21.3**	**19.8**
		time	27.8 + 6.4 = **34.2**	31.1 + 8.0 = **39.1**	41.3 + 8.9 = **50.2**
	no rotations	#its.	**29.0**	**32.8**	**35.5**
		time	26.2 + 9.4 = **35.6**	27.3 + 11.8 = **39.1**	31.1 + 14.3 = **45.4**
	fully algebraic	#its.	**60.7**	**78.5**	**83.0**
		time	27.9 + 19.9 = **47.8**	28.7 + 27.9 = **56.6**	34.1 + 33.1 = **67.2**

4 Newton iterations (with backtracking) were necessary for convergence (relative residual reduction of 10^{-8}) for all configurations.

Computations on **magnitUDE (University Duisburg-Essen)**.

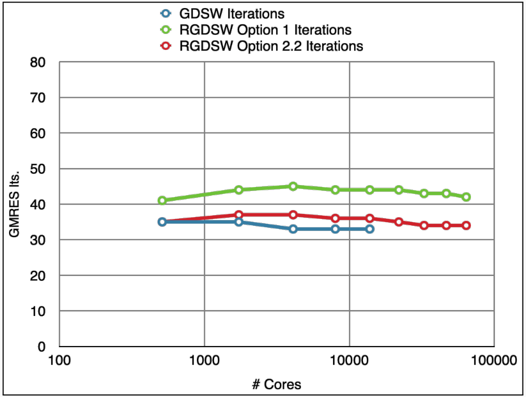
Heinlein, Hochmuth, and Klawonn (2021)

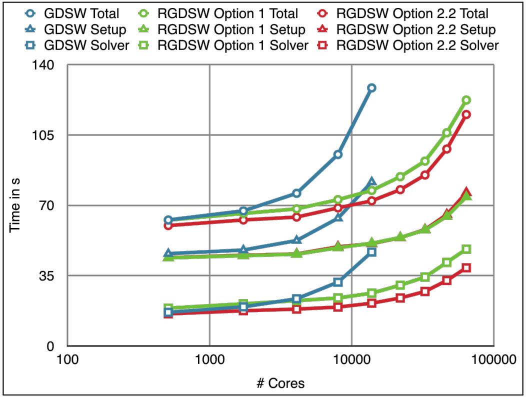
Weak Scalability up to $64\,\mathrm{k}$ MPI ranks / $1.7\,\mathrm{b}$ Unknowns (3D Poisson; Juqueen)

Model problem: **Poisson equation in 3D**

Coarse solver: MUMPS (direct)

Largest problem: **374 805 361 / 1 732 323 601 unknowns**



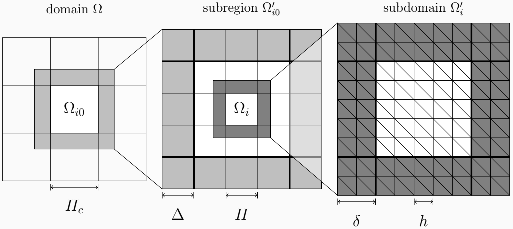


Cf. Heinlein, Klawonn, Rheinbach, Widlund (2017); computations performed on Juqueen, JSC, Germany.

⇒ Using the **reduced dimension coarse space**, we can **improve parallel scalability**.

To **extend the scalability even further**, we consider **multi-level Schwarz preconditioners**.

Three-Level GDSW Preconditioner



Heinlein, Klawonn, Rheinbach, Röver (2019, 2020), Heinlein, Rheinbach, Röver (2022)

Recursive approach

Instead of solving the coarse problem exactly, we apply another GDSW preconditioner on the coarse level ⇒ **recursive application of the GDSW preconditioner**.

Therefore, we introduce **coarse subdomains on the coarse level**, denoted as **subregions**.

The **three-level GDSW preconditioner** is defined as

$$\boldsymbol{M}_{3GDSW}^{-1} = \underbrace{\Phi\Big(\overbrace{\Phi_0 \boldsymbol{K}_{00}^{-1}\Phi_0^T}^{\text{third level}} + \overbrace{\sum\nolimits_{i=1}^{N_0} \boldsymbol{R}_{i0}^T K_{i0}^{-1} \boldsymbol{R}_{i0}}^{\text{second level}}\Big)\Phi^T}_{\textbf{coarse levels}} + \overbrace{\sum\nolimits_{j=1}^{N} \boldsymbol{R}_j^T \boldsymbol{K}_j^{-1} \boldsymbol{R}_j}^{\text{first level}},$$

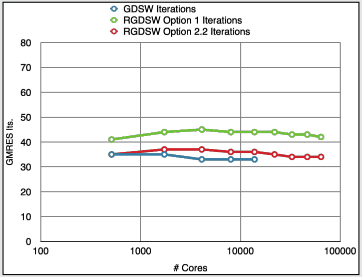
where $K_{00} = \Phi_0^T K_0 \Phi_0$ and $K_{i0} = R_{i0} K_0 R_{i0}^T \quad \text{for } i = 1, \cdots, N_0.$

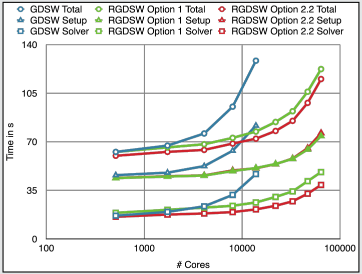
Here, let $R_{i0} : V^0 \to V_i^0 := V^0(\Omega'_{i0})$ for $i = 1, ..., N_0$ be **restriction operators on the subregion level** and Φ_0 contain to corresponding **coarse basis functions**. Our approach is related to other three-level DD methods; cf., e.g., three-level BDDC by **Tu (2007)**.

Weak Scalability up to $64\,\mathrm{k}$ MPI ranks / $1.7\,\mathrm{b}$ Unknowns (3D Poisson; Juqueen)

GDSW vs RGDSW (reduced dimension)

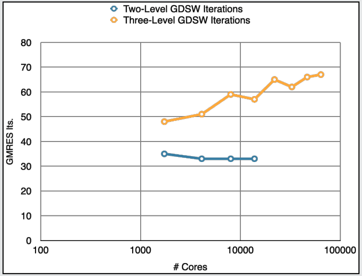
Heinlein, Klawonn, Rheinbach, Widlund (2019).

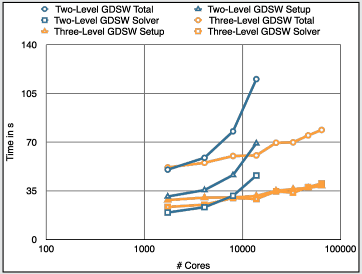




Two-level vs three-level GDSW

Heinlein, Klawonn, Rheinbach, Röver (2019, 2020).

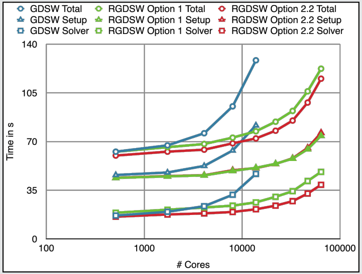




Weak Scalability up to $64\,\mathrm{k}$ MPI ranks / $1.7\,\mathrm{b}$ Unknowns (3D Poisson; Juqueen)

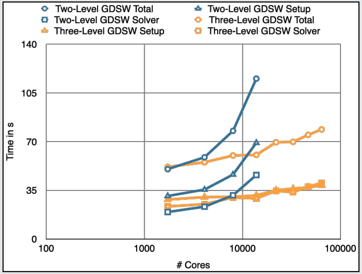
GDSW vs RGDSW (reduced dimension)

Heinlein, Klawonn, Rheinbach, Widlund (2019).



Two-level vs three-level GDSW

Heinlein, Klawonn, Rheinbach, Röver (2019, 2020).



# subdomains (=#cores)		1 728	4 096	8 000	13 824	21 952	32 768	46 656	64 000
GDSW	Size of K_0	10 439	25 695	51 319	89 999	-	-	-	-
	Size of K_{00}	98	279	604	1 115	1 854	2 863	4 184	5 589
RGDSW	Size of K_0	1 331	3 375	6 859	12 167	19 683	29 791	42 875	59 319
	Size of K_{00}	8	27	64	125	216	343	512	729

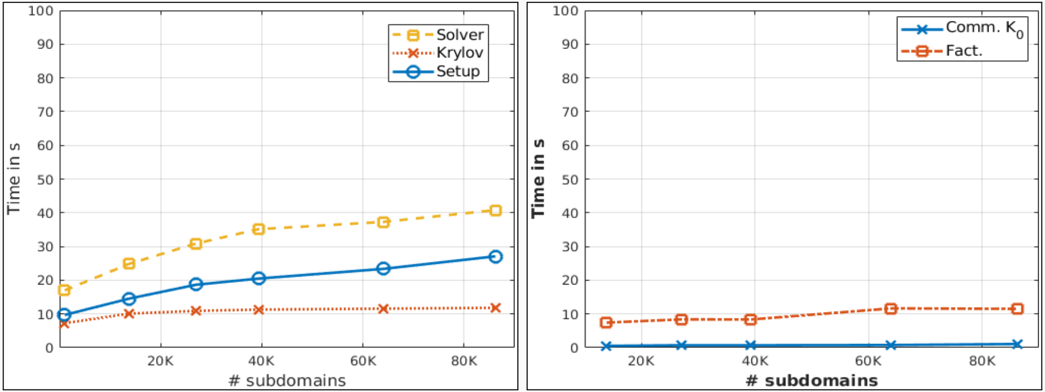
Weak Scalability of the Three-Level RGDSW Preconditioner – SuperMUC-NG

In **Heinlein, Rheinbach, Röver (2022)**, it has been shown that the **null space can be transferred algebraically to higher levels**.

Model problem: **Linear elasticity in 3D**

Largest problem: **2 040 000 000 unknowns**

Coarse solver level 3: Intel MKL Pardiso (direct)



Cf. Heinlein, Rheinbach, Röver (2022); computations performed on SuperMUC-NG, LRZ, Germany.

Weak Scalability of the Three-Level RGDSW Preconditioner – Theta

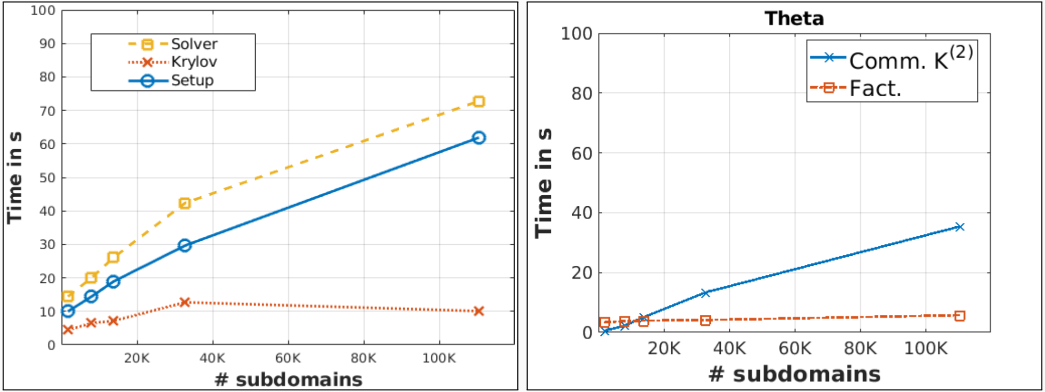
Model problem: **Linear elasticity in 3D**

Largest problem: **1 118 934 000 unknowns**

Coarse solver level 3: Intel MKL Pardiso (direct)

2 OpenMP threads (=cores) per MPI rank

Total: **221 184 cores**



Cf. Heinlein, Rheinbach, Röver (2022); computations performed on Theta, ALCF, USA.

Weak Scalability of the Three-Level RGDSW Preconditioner – Theta

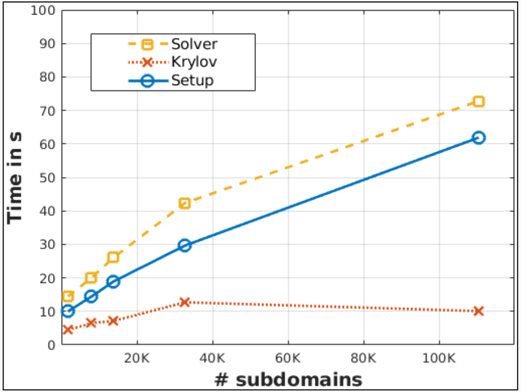
Model problem: **Linear elasticity in 3D**

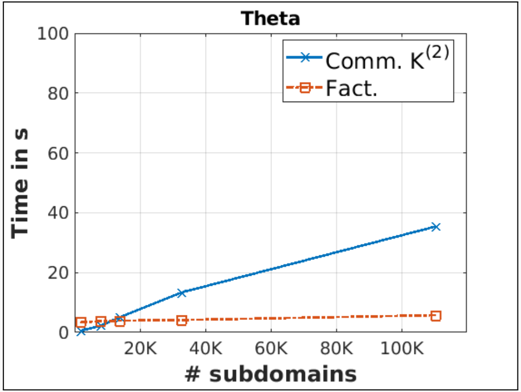
Largest problem: **1 118 934 000 unknowns**

Coarse solver level 3: Intel MKL Pardiso (direct)

2 OpenMP threads (=cores) per MPI rank

Total: **221 184 cores**





Cf. Heinlein, Rheinbach, Röver (2022); computations performed on Theta, ALCF, USA.

Different network topologies of SuperMUC-NG (*fat tree*) and Theta (*Dragonfly*) result in strongly varying communication times.

Monolithic (R)GDSW Preconditioners for CFD Simulations

Monolithic GDSW preconditioner

Consider the discrete saddle point problem

$$\mathcal{A}x = \begin{bmatrix} K & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix} = \mathcal{b}.$$

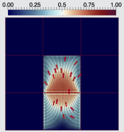
We construct a **monolithic GDSW preconditioner**

$$\mathcal{M}_{\mathrm{GDSW}}^{-1} = \phi \mathcal{A}_0^{-1} \phi^T + \sum_{i=1}^{N} \mathcal{R}_i^T \mathcal{A}_i^{-1} \mathcal{R}_i,$$

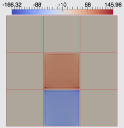
with block matrices $\mathcal{A}_0 = \phi^T \mathcal{A} \phi$, $\mathcal{A}_i = \mathcal{R}_i \mathcal{A} \mathcal{R}_i^T$, and

$$\mathcal{R}_i = \begin{bmatrix} \mathcal{R}_{u,i} & 0 \\ 0 & \mathcal{R}_{p,i} \end{bmatrix} \quad \text{and} \quad \phi = \begin{bmatrix} \Phi_{u,u_0} & \Phi_{u,p_0} \\ \Phi_{p,u_0} & \Phi_{p,p_0} \end{bmatrix}.$$

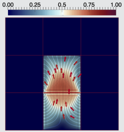
Using $\mathcal{A}$ to compute extensions: $\phi_I = -\mathcal{A}_{II}^{-1} \mathcal{A}_{I\Gamma} \phi_\Gamma$; cf. Heinlein, Hochmuth, Klawonn (2019, 2020).



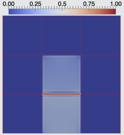
Φ_{u,u_0}



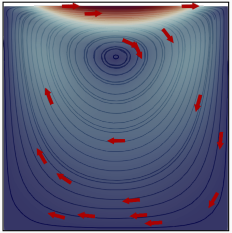
Φ_{p,u_0}



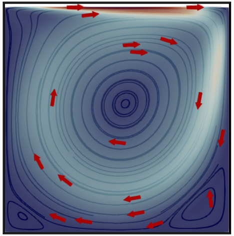
Φ_{u,p_0}



Φ_{p,p_0}



Stokes flow



Navier–Stokes flow

Related work:

- Original work on monolithic Schwarz preconditioners: Klawonn and Pavarino (1998, 2000)
- Other publications on monolithic Schwarz preconditioners: e.g., Hwang and Cai (2006), Barker and Cai (2010), Wu and Cai (2014), and the presentation Dohrmann (2010) at the *Workshop on Adaptive Finite Elements and Domain Decomposition Methods* in Milan.

Monolithic (R)GDSW Preconditioners for CFD Simulations

Monolithic GDSW preconditioner

Consider the discrete saddle point problem

$$\mathcal{A}x = \begin{bmatrix} \boldsymbol{K} & \boldsymbol{B}^T \\ \boldsymbol{B} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{0} \end{bmatrix} = \mathcal{b}.$$

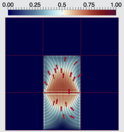
We construct a **monolithic GDSW preconditioner**

$$\mathcal{M}_{\mathrm{GDSW}}^{-1} = \phi \mathcal{A}_0^{-1} \phi^T + \sum\nolimits_{i=1}^{N} \mathcal{R}_i^T \mathcal{A}_i^{-1} \mathcal{R}_i,$$

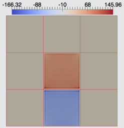
with block matrices $\mathcal{A}_0 = \phi^T \mathcal{A} \phi$, $\mathcal{A}_i = \mathcal{R}_i \mathcal{A} \mathcal{R}_i^T$, and

$$\mathcal{R}_i = \begin{bmatrix} \mathcal{R}_{u,i} & \boldsymbol{0} \\ \boldsymbol{0} & \mathcal{R}_{p,i} \end{bmatrix} \quad \text{and} \quad \phi = \begin{bmatrix} \Phi_{u,u_0} & \Phi_{u,p_0} \\ \Phi_{p,u_0} & \Phi_{p,p_0} \end{bmatrix}.$$

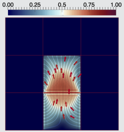
Using $\mathcal{A}$ to compute extensions: $\phi_I = -\mathcal{A}_{II}^{-1} \mathcal{A}_{I\Gamma} \phi_\Gamma$; cf. **Heinlein, Hochmuth, Klawonn (2019, 2020)**.



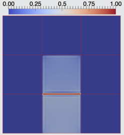
Φ_{u,u_0}



Φ_{p,u_0}

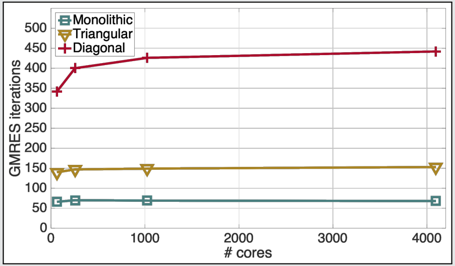


Φ_{u,p_0}



Φ_{p,p_0}

Monolithic vs block preconditioners



prec.	MPI ranks	64	256	1 024	4 096
monolithic	time	154.7 s	170.0 s	175.8 s	188.7 s
	effic.	**100 %**	91 %	88 %	82 %
triangular	time	309.4 s	329.1 s	359.8 s	396.7 s
	effic.	50 %	47 %	43 %	39 %
diagonal	time	736.7 s	859.4 s	966.9 s	1 105.0 s
	effic.	21 %	18 %	16 %	14 %

Computations performed on **magnitUDE (University Duisburg-Essen)**.

Monolithic (R)GDSW Preconditioners for CFD Simulations

Monolithic GDSW preconditioner

Consider the discrete saddle point problem

$$\mathcal{A}x = \begin{bmatrix} \boldsymbol{K} & \boldsymbol{B}^T \\ \boldsymbol{B} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{0} \end{bmatrix} = \mathscr{b}.$$

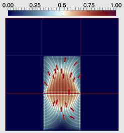
We construct a **monolithic GDSW preconditioner**

$$\mathcal{M}_{\mathrm{GDSW}}^{-1} = \phi \mathcal{A}_0^{-1} \phi^T + \sum\nolimits_{i=1}^{N} \mathcal{R}_i^T \mathcal{A}_i^{-1} \mathcal{R}_i,$$

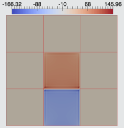
with block matrices $\mathcal{A}_0 = \phi^T \mathcal{A} \phi$, $\mathcal{A}_i = \mathcal{R}_i \mathcal{A} \mathcal{R}_i^T$, and

$$\mathcal{R}_i = \begin{bmatrix} \mathcal{R}_{u,i} & \boldsymbol{0} \\ \boldsymbol{0} & \mathcal{R}_{p,i} \end{bmatrix} \quad \text{and} \quad \phi = \begin{bmatrix} \Phi_{u,u_0} & \Phi_{u,p_0} \\ \Phi_{p,u_0} & \Phi_{p,p_0} \end{bmatrix}.$$

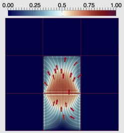
Using $\mathcal{A}$ to compute extensions: $\phi_I = -\mathcal{A}_{II}^{-1} \mathcal{A}_{I\Gamma} \phi_\Gamma$; cf. **Heinlein, Hochmuth, Klawonn (2019, 2020)**.



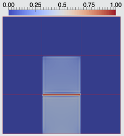
Φ_{u,u_0}



Φ_{p,u_0}

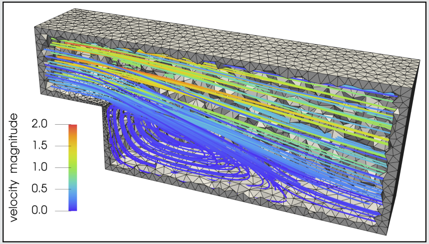


Φ_{u,p_0}



Φ_{p,p_0}

Monolithic vs SIMPLE preconditioner



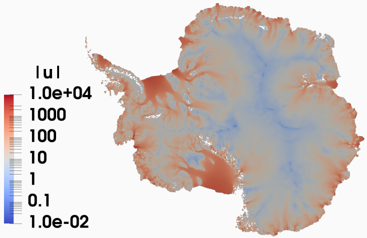
Steady-state Navier-Stokes equations

prec.	MPI ranks	243	1 125	15 562
Monolithic RGDSW (FROSCH)	setup	39.6 s	57.9 s	95.5 s
	solve	57.6 s	69.2 s	74.9 s
	total	**97.2 s**	**127.7 s**	**170.4 s**
SIMPLE RGDSW (Teko & FROSCH)	setup	39.2 s	38.2 s	68.6 s
	solve	86.2 s	106.6 s	127.4 s
	total	125.4 s	144.8 s	196.0 s

Computations on Piz Daint (CSCS). Implementation in the finite element software `FEDDLib`.

FROSch Preconditioners for Land Ice Simulations

https://github.com/SNLComputation/Albany



The velocity of the ice sheet in Antarctica and Greenland is modeled by a **first-order-accurate Stokes approximation model**,

$$-\nabla \cdot (2\mu\dot{\epsilon}_1) + \rho g \frac{\partial s}{\partial x} = 0, \quad -\nabla \cdot (2\mu\dot{\epsilon}_2) + \rho g \frac{\partial s}{\partial y} = 0,$$

with a **nonlinear viscosity model** (Glen's law); cf., e.g., **Blatter (1995)** and **Pattyn (2003)**.

	Antarctica (**velocity**)			Greenland (**multiphysics vel. & temperature**)		
	4 km resolution, 20 layers, 35 m dofs			1-10 km resolution, 20 layers, 69 m dofs		
MPI ranks	avg. its	avg. setup	avg. solve	avg. its	avg. setup	avg. solve
512	41.9 (11)	25.10 s	12.29 s	41.3 (36)	18.78 s	4.99 s
1 024	43.3 (11)	9.18 s	5.85 s	53.0 (29)	8.68 s	4.22 s
2 048	41.4 (11)	4.15 s	2.63 s	62.2 (86)	4.47 s	4.23 s
4 096	41.2 (11)	1.66 s	1.49 s	68.9 (40)	2.52 s	2.86 s
8 192	40.2 (11)	1.26 s	1.06 s	-	-	-

Computations performed on Cori (NERSC).

Heinlein, Perego, Rajamanickam (2022)

Node-Level Performance of FROSch Preconditioners

Inexact Subdomain Solvers in FROSch

$$M_{\text{OS-2}}^{-1} K = \Phi K_0^{-1} \Phi^T K + \sum_{i=1}^{N} R_i^T \mathbf{K}_{\mathbf{i}}^{-1} R_i K$$

3D Laplacian; 512 MPI ranks $=$ 512 ($= 8 \times 8 \times 8$) subdomains; $H/\delta = 10$; RGDSW coarse space.

		subdomain solver						
		direct	ILU(k)		symm. Gauß–Seidel		Chebyshev polyn.	
		solver	k = 2	k = 3	5 sweeps	10 sweeps	p = 6	p = 8
$H/h = 20$, $\approx 14\ k$ dofs per rank	iter	**26**	33	30	31	28	34	31
	setup time	1.89 s	0.97 s	1.01 s	0.89 s	0.91 s	**0.73 s**	**0.71 s**
	apply time	0.39 s	**0.27 s**	0.31 s	0.31 s	0.35 s	0.30 s	0.30 s
	prec. time	2.28 s	1.24 s	1.32 s	1.20 s	1.26 s	1.03 s	**1.01 s**
$H/h = 40$, $\approx 105\ k$ dofs per rank	iter	**30**	55	46	52	41	59	51
	setup time	12.09 s	6.14 s	6.26 s	5.74 s	5.89 s	**5.55 s**	5.64 s
	apply time	4.21 s	**1.84 s**	1.96 s	2.66 s	3.28 s	2.52 s	2.47 s
	prec. time	16.30 s	**7.98 s**	8.22 s	8.40 s	9.18 s	8.16 s	8.11 s
$H/h = 60$, $\approx 350\ k$ dofs per rank	iter	OOM	81	64	76	**56**	88	74
	setup time	-	47.29 s	47.87 s	45.14 s	**45.08 s**	45.44 s	45.49 s
	apply time	-	10.79 s	**9.98 s**	13.00 s	16.16 s	11.95 s	12.09 s
	prec. time	-	58.08 s	57.85 s	58.15 s	61.25 s	**57.39 s**	57.59 s

INTEL MKL PARDISO; ILU / symmetric Gauß–Seidel / Chebyshev polynomials from IFPACK2.

Parallel computations on dual-socket Intel Xeon Platinum machine at Sandia National Laboratories (Blake).

Inexact Subdomain Solvers in FROSch

$$\boldsymbol{M}_{\text{OS-2}}^{-1}\boldsymbol{K} = \Phi \boldsymbol{K}_0^{-1}\Phi^T\boldsymbol{K} + \sum_{i=1}^{N} \boldsymbol{R}_i^T \mathbf{K}_{\mathbf{i}}^{-\mathbf{1}} \boldsymbol{R}_i \boldsymbol{K}$$

3D Laplacian; 512 MPI ranks = 512 (= 8 × 8 × 8) subdomains; $H/\delta = 10$; RGDSW coarse space.

		subdomain solver						
		direct	ILU(k)		symm. Gauß–Seidel		Chebyshev polyn.	
		solver	k = 2	k = 3	5 sweeps	10 sweeps	p = 6	p = 8
$H/h = 20$, ≈ 14 k dofs per rank	iter	**26**	33	30	31	28	34	31
	setup time	1.89 s	0.97 s	1.01 s	0.89 s	0.91 s	**0.73 s**	**0.71 s**
	apply time	0.39 s	**0.27 s**	0.31 s	0.31 s	0.35 s	0.30 s	0.30 s
	prec. time	2.28 s	1.24 s	1.32 s	1.20 s	1.26 s	1.03 s	**1.01 s**
$H/h = 40$, ≈ 105 k dofs per rank	iter	**30**	55	46	52	41	59	51
	setup time	12.09 s	6.14 s	6.26 s	5.74 s	5.89 s	**5.55 s**	5.64 s
	apply time	4.21 s	**1.84 s**	1.96 s	2.66 s	3.28 s	2.52 s	2.47 s
	prec. time	16.30 s	**7.98 s**	8.22 s	8.40 s	9.18 s	8.16 s	8.11 s
$H/h = 60$, ≈ 350 k dofs per rank	iter	OOM	81	64	76	**56**	88	74
	setup time	-	47.29 s	47.87 s	45.14 s	**45.08 s**	45.44 s	45.49 s
	apply time	-	10.79 s	**9.98 s**	13.00 s	16.16 s	11.95 s	12.09 s
	prec. time	-	58.08 s	57.85 s	58.15 s	61.25 s	**57.39 s**	57.59 s

Intel MKL Pardiso; ILU / symmetric Gauß–Seidel / Chebyshev polynomials from Ifpack2.

Parallel computations on dual-socket Intel Xeon Platinum machine at Sandia National Laboratories (Blake).

Inexact Extension Solvers in FROSch

$$\Phi = \begin{bmatrix} -\mathbf{K}_{\mathbf{II}}^{-1} K_{\Gamma I}^{T} \Phi_\Gamma \\ \Phi_\Gamma \end{bmatrix} = \begin{bmatrix} \Phi_I \\ \Phi_\Gamma \end{bmatrix}.$$

3D Laplacian; 512 MPI ranks = 512 (= 8 × 8 × 8) subdomains; $H/\delta = 10$; RGDSW coarse space.

extension solver (10 Gauss–Seidel sweeps for the subdomain solver)		direct solver	**preconditioned GMRES** (rel. tol. $= 10^{-4}$)					
			ILU(k)		symm. Gauß–Seidel		Chebyshev polyn.	
			k = 2	k = 3	5 sweeps	10 sweeps	p = 6	p = 8
$H/h = 20$, $\approx 14\,k$ dofs per rank	iter	**28**	**28**	**28**	**28**	**28**	**28**	**28**
	setup time	0.89 s	0.93 s	0.89 s	**0.78 s**	0.83 s	0.79 s	0.84 s
	apply time	0.35 s	0.35 s	**0.34 s**	0.36 s	**0.34 s**	0.35 s	**0.34 s**
	prec. time	1.23 s	1.28 s	1.23 s	**1.14 s**	1.17 s	**1.14 s**	1.18 s
$H/h = 40$, $\approx 105\,k$ dofs per rank	iter	**41**	**41**	**41**	**41**	**41**	**41**	**41**
	setup time	5.72 s	**4.16 s**	4.61 s	4.26 s	4.64 s	4.27 s	4.33 s
	apply time	3.33 s	3.33 s	3.30 s	3.33 s	3.30 s	**3.28 s**	3.29 s
	prec. time	9.04 s	**7.49 s**	7.92 s	7.59 s	7.95 s	7.55 s	7.62 s
$H/h = 60$, $\approx 350\,k$ dofs per rank	iter	**56**	**56**	**56**	**56**	**56**	**56**	**56**
	setup time	45.16 s	**17.75 s**	18.16 s	17.98 s	19.34 s	17.93 s	18.04 s
	apply time	**15.83 s**	18.04 s	17.08 s	16.26 s	15.81 s	16.19 s	16.44 s
	prec. time	60.99 s	35.79 s	35.25 s	34.24 s	35.15 s	**34.12 s**	34.49 s

Intel MKL Pardiso; ILU / symmetric Gauß–Seidel / Chebyshev polynomials from Ifpack2.

Parallel computations on dual-socket Intel Xeon Platinum machine at Sandia National Laboratories (Blake).

Inexact Extension Solvers in FROSch

$$\Phi = \begin{bmatrix} -\mathbf{K}_{II}^{-1} K_{\Gamma I}^{T} \Phi_{\Gamma} \\ \Phi_{\Gamma} \end{bmatrix} = \begin{bmatrix} \Phi_{I} \\ \Phi_{\Gamma} \end{bmatrix}.$$

3D Laplacian; 512 MPI ranks = 512 (= 8 × 8 × 8) subdomains; $H/\delta = 10$; RGDSW coarse space.

extension solver (10 Gauss–Seidel sweeps for the subdomain solver)		direct solver	**preconditioned GMRES** (rel. tol. = 10^{-4})					
			ILU(k)		symm. Gauß–Seidel		Chebyshev polyn.	
			k = 2	k = 3	5 sweeps	10 sweeps	p = 6	p = 8
$H/h = 20$, $\approx 14\ k$ dofs per rank	iter	**28**	**28**	**28**	**28**	**28**	**28**	**28**
	setup time	0.89 s	0.93 s	0.89 s	**0.78 s**	0.83 s	0.79 s	0.84 s
	apply time	0.35 s	0.35 s	**0.34 s**	0.36 s	**0.34 s**	0.35 s	**0.34 s**
	prec. time	1.23 s	1.28 s	1.23 s	**1.14 s**	1.17 s	**1.14 s**	1.18 s
$H/h = 40$, $\approx 105\ k$ dofs per rank	iter	**41**	**41**	**41**	**41**	**41**	**41**	**41**
	setup time	5.72 s	**4.16 s**	4.61 s	4.26 s	4.64 s	4.27 s	4.33 s
	apply time	3.33 s	3.33 s	3.30 s	3.33 s	3.30 s	**3.28 s**	3.29 s
	prec. time	9.04 s	**7.49 s**	7.92 s	7.59 s	7.95 s	7.55 s	7.62 s
$H/h = 60$, $\approx 350\ k$ dofs per rank	iter	**56**	**56**	**56**	**56**	**56**	**56**	**56**
	setup time	45.16 s	**17.75 s**	18.16 s	17.98 s	19.34 s	17.93 s	18.04 s
	apply time	**15.83 s**	18.04 s	17.08 s	16.26 s	15.81 s	16.19 s	16.44 s
	prec. time	60.99 s	35.79 s	35.25 s	34.24 s	35.15 s	**34.12 s**	34.49 s

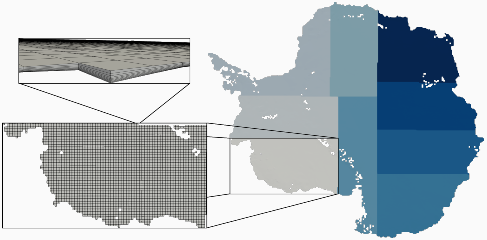
Intel MKL Pardiso; ILU / symmetric Gauß–Seidel / Chebyshev polynomials from Ifpack2.

Parallel computations on dual-socket Intel Xeon Platinum machine at Sandia National Laboratories (Blake).

Land Ice Problem – OpenMP VS MPI Parallelization (Strong Scaling)

We can make use of **OpenMP parallelization**:

- TPETRA linear algebra stack in FROSCH and ALBANY ⇒ **OpenMP parallelization of the linear algebra operations**.
- **OpenMP parallelization** of the **subdomain and coarse solver** INTEL MKL PARDISO used in FROSCH.



Antarctica mesh & domain decomposition.

	OpenMP parallelization (512 MPI ranks)				**MPI parallelization**			
cores	OpenMP threads	avg. its (nl its)	avg. setup	avg. solve	MPI ranks	avg. its (nl its)	avg. setup	avg. its solve
512	1	**42.6** (11)	**14.99 s**	**12.50 s**	512	**42.6** (11)	**14.99 s**	**12.50 s**
1 024	2	**42.6** (11)	9.43 s	6.80 s	1 024	44.5 (11)	**5.65 s**	**6.08 s**
2 048	4	**42.6** (11)	5.50 s	4.02 s	2 048	42.7 (11)	**3.11 s**	**2.79 s**
4 096	8	42.6 (11)	3.65 s	2.71 s	4 096	**42.5** (11)	**1.07 s**	**1.54 s**
8 192	16	42.6 (11)	2.56 s	2.32 s	8 192	**42.0** (11)	**1.20 s**	**1.16 s**

Problem:	Velocity	**Mesh:**	Antarctica 4 km hor. resolution 20 vert. layers	**Size:**	35.3 m degrees of freedom (P1 FE)	**Coarse space:**	RGDSW

Sparse Triangular Solver in Kokkos-Kernels (Amesos2 – SuperLU/Cholmod)

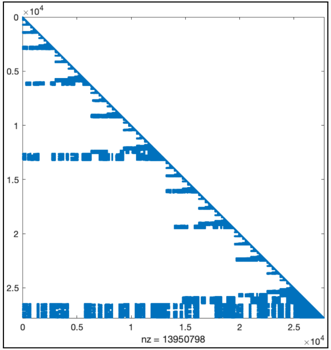
The sparse triangular solver is an **important kernel** in many codes (including `FROSch`) but is **challenging to parallelize**

- Factorization using a **sparse direct solver** typically leads to triangular matrices with **dense blocks** called **supernodes**
- In **supernodal triangular solver**, rows/columns with a similar sparsity pattern are merged into a supernodal block, and the **solve is then performed block-wise**
- The **parallelization potential** for the triangular solver is **determined by the sparsity pattern**

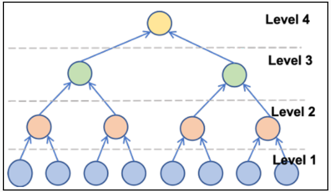
Parallel supernode-based triangular solver:

1. **Supernode-based level-set scheduling**, where **all leaf-supernodes within one level are solved in parallel** (batched kernels for hierarchical parallelism)
2. **Partitioned inverse** of the submatrix associated with each level: **SpTRSV is transformed into a sequence of SpMVs**

See Yamazaki, Rajamanickam, Ellingwood (2020) for more details.



Lower-triangular matrix – SuperLU with METIS nested dissection ordering



Three-Dimensional Linear Elasticity – Weak Scalability

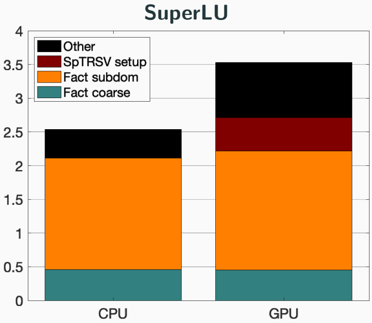
# nodes	1	2	4	8	16
# dofs	375K	750K	1.5M	3M	6M

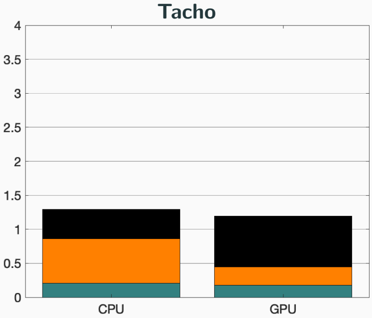
SuperLU					
CPUs	**2.03 (75)**	**2.07 (69)**	**1.87 (61)**	**1.95 (58)**	**2.48 (69)**
n_p/gpu = 1	1.43 (47)	1.52 (53)	2.82 (77)	2.44 (68)	2.61 (75)
2	1.03 (46)	1.36 (65)	1.37 (60)	1.52 (65)	1.98 (86)
4	0.93 (59)	0.91 (53)	0.98 (59)	1.33 (77)	1.21 (66)
6	0.67 (46)	0.99 (65)	0.92 (57)	0.91 (57)	0.95 (57)
7	**1.03 (75)**	**1.04 (69)**	**0.90 (61)**	**0.97 (58)**	**1.18 (69)**
Speedup	**2.0**×	**2.0**×	**2.1**×	**2.0**×	**2.1**×

Tacho					
CPUs	**1.60 (75)**	**1.63 (69)**	**1.49 (61)**	**1.51 (58)**	**1.90 (69)**
n_p/gpu = 1	1.17 (47)	1.37 (53)	1.92 (77)	1.78 (68)	2.21 (75)
2	0.79 (46)	1.14 (65)	1.05 (60)	1.18 (65)	1.70 (86)
4	0.85 (59)	0.81 (53)	0.78 (59)	1.22 (77)	1.19 (66)
6	0.60 (46)	0.86 (65)	0.75 (57)	0.84 (57)	0.91 (57)
7	**0.99 (75)**	**0.93 (69)**	**0.82 (61)**	**0.93 (58)**	**1.22 (69)**
Speedup	**1.6**×	**1.8**×	**1.8**×	**1.6**×	**1.6**×

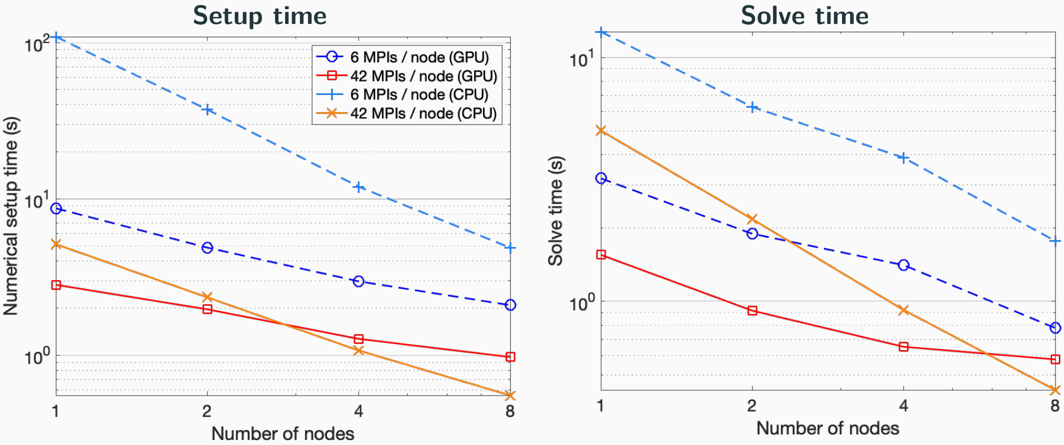
Computations on Summit (OLCF); 42 IBM Power9 CPU cores and 6 NVIDIA V100 GPUs per node.

Yamazaki, Heinlein, Rajamanickam (acc. 2022)





Three-Dimensional Linear Elasticity – Strong Scalability Using Tacho



Cf. **Yamazaki, Heinlein, Rajamanickam (acc. 2022)**

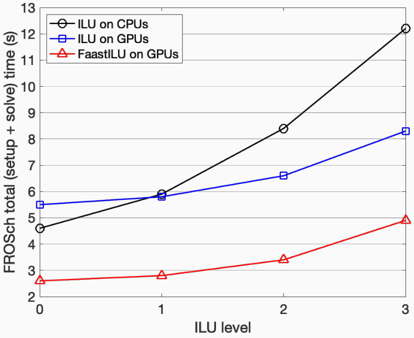
Three-Dimensional Linear Elasticity – ILU Subdomain Solver

ILU level		0	1	2	3
Setup					
CPU	No	**1.5**	**1.9**	**3.0**	**4.8**
	ND	1.6	2.6	4.4	7.4
GPU	KK(No)	1.4	1.5	1.8	2.4
	KK(ND)	1.7	2.0	2.9	5.2
	Fast(No)	**1.5**	**1.6**	**2.1**	**3.2**
	Fast(ND)	1.5	1.7	2.5	4.5
Speedup		**1.0**×	**1.2**×	**1.4**×	**1.5**×
Solve					
CPU	No	**2.55 (158)**	**3.60 (112)**	**5.28 (99)**	**6.85 (88)**
	ND	4.17 (227)	5.36 (134)	6.61 (105)	7.68 (88)
GPU	KK(No)	3.81 (158)	4.12 (112)	4.77 (99)	5.65 (88)
	KK(ND)	2.89 (227)	4.27 (134)	5.57 (105)	6.36 (88)
	Fast(No)	**1.14 (173)**	**1.11 (141)**	**1.26 (134)**	**1.43 (126)**
	Fast(ND)	1.49 (227)	1.15 (137)	1.10 (109)	1.22 (100)
Speedup		**2.2**×	**3.2**×	**4.3**×	**4.8**×

Computations on Summit (OLCF);
42 IBM Power9 CPU cores and 6
NVIDIA V100 GPUs per node.

Yamazaki, Heinlein, Rajamanickam (acc. 2022)

KokkosKernels ILU (KK)
vs
`FastILU` (Fast); cf. **Chow, Patel (2015)**
and **Boman, Patel, Chow, Rajamanickam (2016)**



Three-Dimensional Linear Elasticity – Weak Scalability Using ILU

# nodes		1	2	4	8	16
# dofs		648K	1.2M	2.6M	5.2M	10.3M
Setup						
CPU		**1.9**	**2.2**	**2.4**	**2.4**	**2.6**
GPU	KK	1.4	2.0	2.2	2.4	2.8
	Fast	**1.5**	**2.2**	**2.3**	**2.5**	**2.8**
Speedup		**1.3**×	**1.0**×	**1.0**×	**1.0**×	**0.9**×
Solve						
CPU		**3.60 (112)**	**7.26 (84)**	**6.93 (78)**	**6.41 (75)**	**4.1 (109)**
GPU	KK	4.12 (112)	6.17 (84)	5.82 (78)	5.95 (75)	7.16 (83)
	Fast	**1.11 (141)**	**1.12 (91)**	**1.08 (81)**	**1.21 (76)**	
Speedup		**3.3**×	**3.8**×	**3.4**×	**2.5**×	**2.6**×

Computations on Summit (OLCF);
42 IBM Power9 CPU cores and 6 NVIDIA V100 GPUs per node.

Yamazaki, Heinlein, Rajamanickam (acc. 2022)

Thank you for your attention!

Summary

- Making numerical software ready for the first supercomputers of the Exascale era requires **dealing with heterogeneous computing architectures**. TRILINOS enables this due to the **TPETRA parallel linear algebra framework** as well as **tight integration of the performance portability framework** KOKKOS and KOKKOSKERNELS
- FROSCH is based on the **Schwarz framework** and **energy-minimizing coarse spaces**, which provide **numerical scalability** using **only algebraic information** for a **variety of applications**
- FROSCH is well-integrated into the TRILINOS software framework, enabling
  - **large-scale distributed memory parallelization** and
  - **node-level performance on CPU and/or GPU architectures**

Acknowledgements

- **Financial support:** DFG (KL2094/3-1, RH122/4-1)
- **Computing ressources:** Summit (OLCF), Cori (NERSC), JUQUEEN (JSC), magnitUDE (UDE), Piz Daint (CSCS)