Parallel Scalability of Three-Level FROSch Preconditioners to 220000 Cores using the Theta Supercomputer

Heinlein, Alexander; Rheinbach, Oliver; Röver, Friederike

DOI
10.1137/21M1431205

Publication date
2022

Document Version
Final published version

Published in
SIAM Journal on Scientific Computing

SIAM J. SCI. COMPUT.
Vol. 0, No. 0, pp. S173–S198

PARALLEL SCALABILITY OF THREE-LEVEL FROSch PRECONDITIONERS TO 220000 CORES USING THE THETA SUPERCOMPUTER[*]

ALEXANDER HEINLEIN[†], OLIVER RHEINBACH[‡], AND FRIEDERIKE RÖVER[‡]

Abstract. The parallel performance of the three-level fast and robust overlapping Schwarz (FROSch) preconditioners is investigated for linear elasticity. The FROSch framework is part of the Trilinos software library and contains a parallel implementation of different preconditioners with energy minimizing coarse spaces of generalized Dryja–Smith–Widlund type. The three-level extension is constructed by a recursive application of the FROSch preconditioner to the coarse problem. In this paper, the additional steps in the implementation in order to apply the FROSch preconditioner recursively are described in detail. Furthermore, it is shown that no explicit geometric information is needed in the recursive application of the preconditioner. In particular, the rigid body modes, including the rotations, can be interpolated on the coarse level without additional geometric information. Parallel results for a three-dimensional linear elasticity problem obtained on the Theta supercomputer (Argonne Leadership Computing Facility, Argonne, IL) using up to 220 000 cores are discussed and compared to results obtained on the SuperMUC-NG supercomputer (Leibniz Supercomputing Centre, Garching, Germany). Notably, it can be observed that a hierarchical communication operation in FROSch related to the coarse operator starts to dominate the computing time on Theta, which has a dragonfly interconnect, for 100 000 message passing interface (MPI) ranks or more. The same operation, however, scales well and stays within the order of a second in all experiments performed on SuperMUC-NG, which uses a fat tree network. Using hybrid MPI/OpenMP parallelization, the onset of the MPI communication problem on Theta can be delayed. Further analysis of the performance of FROSch on large supercomputers with dragonfly interconnects will be necessary.

Key words. domain decomposition, high performance computing, overlapping Schwarz, software, Trilinos, multilevel preconditioners

MSC codes. 65N55, 74B05

DOI. 10.1137/21M1431205

1. Introduction.

We consider the parallel scalability of recent three-level overlapping Schwarz domain decomposition preconditioners implemented in the Fast and robust overlapping Schwarz (FROSch) software [19, 18], which is part of the Trilinos software library [2].

Since the use of a direct coarse solver limits the parallel scalability of two-level methods, we focus on using three-level methods where the coarse problem is solved inexactly by applying another two-level preconditioner. Hence, these approaches are based on the recursive construction of two-level preconditioners. In particular, we

[*]Received by the editors July 7, 2021; accepted for publication (in revised form) April 22, 2022; published electronically August 23, 2022.
https://doi.org/10.1137/21M1431205

Funding: This work was supported by the Gauss Centre for Supercomputing and by the DFG through grant 397252409. The second and third author would like to acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG) under DFG project 441509557 within the DFG SPP 2256 "Variational Methods for Predicting Complex Phenomena in Engineering Structures and Materials."

[†]Delft University of Technology, Faculty of Electrical Engineering Mathematics & Computer Science, Delft Institute of Applied Mathematics, Mekelweg 4, 2628 CD Delft, Netherlands (a.heinlein@tudelft.nl).

[‡]Fakultät für Mathematik und Informatik, Zentrum für effiziente Hochtemperatur-Stoffwandlung (ZeHS), and Universitätsrechenzentrum (URZ), Technische Universität Bergakademie Freiberg, 09596 Freiberg, Germany (oliver.rheinbach@math.tu-freiberg.de, roeverf@unixmail.tu-freiberg.de).

will use Schwarz preconditioners with energy minimizing coarse spaces which, once a coarse level is available, can be constructed recursively in a fully algebraic way. These methods are known as generalized Dryja–Smith–Wildund (GDSW) [12, 11] and reduced dimension generalized Dryja–Smith–Wildund (RGDSW) [13, 23] type preconditioners, and they are well-suited for an efficient three-level implementation. In [20, 21], three-level GDSW/RGDSW preconditioners have been introduced for two- and three-dimensional problems, and first results have been presented. Furthermore, in [25], the partitioning of the coarse problem has been discussed.

Here, we discuss the three-level implementation in FROSch in detail and show parallel results for scalability up to 220 000 cores on the Argonne Leadership Computing Facility (ALCF; Argonne, IL) Theta supercomputer, which is facilitated by the object-oriented FROSch implementation of three-level GDSW/RGDSW preconditioners. The computational results on Theta were obtained during the ALCF Performance Workshop 2021.

We provide a comparison with computational results previously obtained on up to 85 000 cores of the SuperMUC-NG supercomputer; cf. [22] for a detailed discussion of these results. We also discuss that, on Theta, possibly due to message passing interface (MPI) congestion, using hybrid MPI/OpenMP parallelization can delay the communication problem compared to pure MPI. The hybrid parallelization uses the MPI parallel linear algebra provided by Tpetra and the shared memory parallel linear algebra kernels from Kokkos Kernels through the Kokkos programming model [9]. Moreover, we use the shared memory parallelization of the sparse direct solver PardisoMKL [7] used for the subdomain problems and the coarsest-level problem. Hybrid parallelization for two-level FROSch preconditioners has previously been successfully used for large scale land ice simulations for Antarctica and Greenland; see [24].

Our implementation is related to other parallel implementations of scalable overlapping Schwarz methods [26, 31]. The three-level GDSW and RGDSW approaches are also related to three-level (or multilevel) balancing domain decomposition by constraints (BDDC) methods [35, 30, 4, 33].

2. FROSch preconditioners.

The FROSch framework [19, 18] contains a parallel implementation of the GDSW preconditioner [11]. The GDSW preconditioner is a two-level overlapping Schwarz domain decomposition preconditioner [34] with an energy minimizing coarse space and a condition number bound.

An important feature of the GDSW coarse space is that it can be constructed algebraically from the fully assembled stiffness matrix; in particular, it requires neither a coarse triangulation nor Neumann matrices for the subdomains. Note that, in this paper, for linear elasticity, we assume that a basis of the nullspace of the operator (or approximations thereof) can be provided to the preconditioner by the user. Neglecting certain dimensions of the coarse space, e.g., for elasticity the linearized rotations, which cannot be constructed algebraically, GDSW may still scale [19, 17], but this is not covered by theory.

For the construction of the preconditioner, the computational domain is first decomposed into N nonoverlapping subdomains $\{\Omega_i\}_{i=1,\dots,N}$. Then, each subdomain is extended by k layers of elements in order to obtain overlapping subdomains $\{\Omega_i'\}_{i=1,\dots,N}$ with an overlap $\delta = kh$, forming the overlapping domain decomposition of the first level. Furthermore, let V be the global finite element space, V_i the local finite element space on the overlapping subdomain Ω_i', and R_i the restriction operator restricting functions from V to V_i.

Let $M_{\mathrm{GDSW}}^{-1}K$ be the preconditioned system matrix. For the theory, we assume that the system matrix K is symmetric positive definite. The two-level GDSW preconditioner can be written in the following form:

$$M_{\mathrm{GDSW}}^{-1} = \underbrace{\Phi K_0^{-1} \Phi^T}_{\text{coarse level}} + \underbrace{\sum\nolimits_{i=1}^{N} R_i^T K_i^{-1} R_i}_{\text{first level}}, \tag{2.1}$$

where $K_i = R_i K R_i^T, i = 1, \dots N$, are local subdomain matrices on the overlapping subdomains. For the coarse level, we have $K_0 = \Phi^T K \Phi$. The matrix Φ contains the coarse basis functions as columns, which span the coarse space V_0. In the classical approach, these would be nodal finite element functions on a coarse triangulation. In GDSW, for the coarse space, certain interface functions Φ_Γ are extended to the interior of each subdomain in an energy minimizing way. In particular,

$$\Phi = \begin{pmatrix} \Phi_I \\ \Phi_\Gamma \end{pmatrix} = \begin{pmatrix} -K_{II}^{-1} K_{I\Gamma} \Phi_\Gamma \\ \Phi_\Gamma \end{pmatrix}, \tag{2.2}$$

where K_{II} and $K_{I\Gamma}$ are submatrices of K if ordered with respect to the interior (I) and interface (Γ) degrees of freedom,

$$K = \begin{pmatrix} K_{II} & K_{I\Gamma} \\ K_{\Gamma I} & K_{\Gamma\Gamma} \end{pmatrix}.$$

Note that $K_{II} = \mathrm{diag}(K_{II}^{(i)})$ is a block-diagonal matrix, where $K_{II}^{(i)}$ corresponds to the ith nonoverlapping subdomain. Therefore, the extensions can be computed independently and concurrently for all subdomains. The choice of the functions Φ_Γ is different for GDSW and the (more recent) RGDSW methods. In GDSW methods, the interface functions Φ_Γ are restrictions of the nullspace Z of the global Neumann matrix to the vertices, edges, and faces, which form a nonoverlapping decomposition of the interface Γ of the nonoverlapping domain decomposition.

In detail, the columns of Φ_Γ are

$$\Phi_\Gamma = \begin{pmatrix} \phi_1^1 & \cdots & \phi_{n_1}^1 & \quad \cdots \quad & \phi_1^k & \cdots & \phi_{n_k}^k \end{pmatrix}, \tag{2.3}$$

where k is the dimension of nullspace Z, where we denote the number of coarse basis functions belonging to the jth basis vector of Z by n_j; the different coarse basis functions correspond to different interface components. Note that the number of coarse basis functions is not necessarily the same for each basis vector of Z. In subsection 5.5, we discuss this in detail. Our interface functions provide a partition of the nullspace restricted to the interface.

Under certain regularity conditions for the domain decomposition and for scalar elliptic and linear elasticity problems, the condition number bound is given by

$$\kappa(M_{\mathrm{GDSW}}^{-1}K) \leq C \left(1 + \frac{H}{\delta}\right)\left(1 + \log\left(\frac{H}{h}\right)\right), \tag{2.4}$$

where h and H are the sizes of the finite elements and the nonoverlapping subdomains, respectively, and C is a constant independent of the other parameters; cf. [12, 11].

Compared to typical coarse spaces of scalable finite element tearing and interconnecting–dual-primal (FETI-DP) [28] and BDDC [34] methods the standard GDSW coarse space is larger, especially in 3D. BDDC and FETI-DP preconditioners, however, cannot be constructed algebraically. A large dimension of the coarse space will lead to a scaling bottleneck if the coarse problem is solved using a direct solver. The parallel scalability of the GDSW method can be extended by using more recent RGDSW coarse spaces [13], which are also implemented in FROSch; cf. [23]. Therefore, for large parallel computations using two-level methods, we often focus on the use of RGDSW coarse spaces. For the RGDSW coarse spaces, we consider a different partition of unity of the interface. The RGDSW coarse space is a nodal coarse space, where the basis functions are associated with the vertices. As in the classical GDSW coarse spaces, we use energy-minimizing extensions to define the interior degrees of freedom; see also Figure 10, where the basis functions are given for linear elasticity. FROSch currently implements two options of RGDSW coarse spaces, the algebraic variant (Option 1), where the interface values are defined based on adjacency relations of interface components, and the geometric variant (Option 2.2) [13, 23], where the interface values are computed using Euclidean distances of interface nodes; see [13, 23] for more details.

Two-level GDSW preconditioners are typically suitable for thousands for cores, while two-level RGDSW preconditioners can scale to tens of thousands of cores. For large numbers of subdomains, however, the coarse problem of both methods may become too large to be solved by a sparse direct linear solver. Therefore, we will here focus on extending the scalability by means of using another level of an (R)GDSW preconditioner as an inexact coarse solver.

3. Three-level extension.

To overcome the scaling bottleneck arising from using a sequential or parallel direct solver on the second level of the GDSW preconditioner, we apply the GDSW preconditioner recursively to the coarse problem [20, 21]. A similar approach is also used in other multilevel domain decomposition methods [35, 30, 4, 33, 29, 32, 31] and, of course, in multigrid methods [16]. Due to the algebraicity of GDSW coarse spaces, a multilevel extension of the GDSW preconditioner may seem relatively straightforward; however, in this paper, we will only investigate the three-level version since it is sufficient for the range of scalability considered here.

For the additional level in the three-level extension, we need to define an additional decomposition of our computational domain Ω. In particular, we decompose Ω into nonoverlapping subregions Ω_{i0} of diameter H_c, where each subregion is a union of nonoverlapping subdomains. In order to obtain an overlapping decomposition into overlapping subregions Ω'_{i0}, we add j layers of subdomains to each subregion, as we do with finite elements on the subdomain level; we denote the subregion overlap by $\Delta = jH$; see Figure 1 for a graphical representation using a structured geometric decompositions into subdomains and subregions. Note that the parallel computational results presented in this paper in section 6 use a geometric decomposition into nonoverlapping subdomains. Alternatively, as investigated computationally in [17, 22], a fully algebraic decomposition can be used to determine the nonoverlapping domain decomposition.

As we will discuss in subsection 5.1, in both cases, the decomposition into subregions is then performed in the same way, based on the decomposition into nonoverlapping subdomains.

With the aforementioned subregions and restriction operators R_{i0} from the coarse space on the second level V_0 to the subspaces V_{i0} on the overlapping subregions Ω'_{i0}, we define the three-level GDSW preconditioner [20, 21]:

$$(3.1)\quad M^{-1}_{\mathrm{GDSW-3L}} = \Phi\Big(\underbrace{\Phi_0 K_{00}^{-1}\Phi_0^T}_{\text{third level}} + \underbrace{\sum\nolimits_{i=1}^{N_0} R_{i0}^T K_{i0}^{-1} R_{i0}}_{\text{second level}}\Big)\Phi^T + \underbrace{\sum\nolimits_{j=1}^{N} R_j^T K_j^{-1} R_j}_{\text{first level}},$$

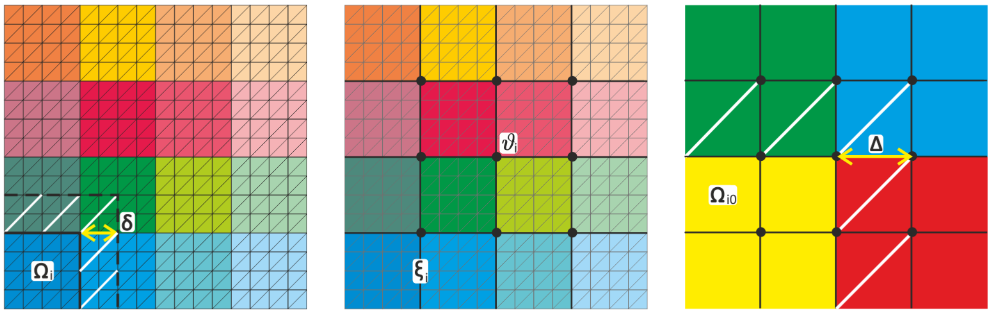

FIG. 1. *Structured decomposition of a two-dimensional computational domain Ω into nonoverlapping subdomains Ω_i (left), corresponding to the first level of our preconditioner. Coarse grid with vertices ϑ_i and edges ξ_i (middle). On the coarse grid, the vertices ϑ_i and edges ξ_i correspond the finite element nodes on the first level. Structured decomposition into nonoverlapping subregions Ω_{i0} (right) corresponding to the coarse level of our preconditioner.*

which, as previously mentioned, arises from replacing K_0^{-1} in (2.1) by another GDSW or RGDSW preconditioner. Therefore, the first level and the matrices Φ are defined as in the two-level method; cf. (2.1). Moreover, the $K_{i0} = R_{i0} K_0 R_{i0}^T$ are the local matrices on the subregions, and $K_{00} = \Phi_0^T K_0 \Phi_0$ is the coarse matrix corresponding to the coarse space on the third level V_{00}, which is spanned by the coarse basis functions Φ_0. The coarse basis functions are constructed as described in section 2 for the two-level case, however, on the subregion level.

4. Model problem.

FROSch is applied in the DFG SPP 2256 as a solver in nonlinear thermo-chemo-mechanics [27], where it is important that the preconditioner can be applied in an algebraic way.

Here, we explore the scalability limits of our three-level preconditioner and its implementation in FROSch, as described in the previous section 5, using the linear elasticity benchmark model problem: find $u \in (H^1(\Omega))^3$ such that

$$\begin{aligned} \operatorname{div} u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega_D, \end{aligned} \tag{4.1}$$

on $\Omega := [0,1]^3$ using a homogenous Dirichlet boundary condition on $\partial\Omega_D := \partial\Omega$.

The Trilinos package Galeri is used to assemble the stiffness matrix. The implementation in the Galeri package is based on trilinear finite elements and a structured decomposition of the computational domain Ω. Instead of specifying a right-hand-side function f, we use the generic discrete right-hand-side vector $(1,\dots,1)^T$ for the resulting linear equation system.

5. Implementation.

In this section, we will describe the extension of the FROSch [19, 18] framework to three-level (R)GDSW preconditioners. FROSch is part of the Trilinos [2] software library and is built on top of the Xpetra package [36], which is a lightweight interface to both Trilinos linear algebra frameworks Epetra and Tpetra. Moreover, FROSch can be easily called through the unified Stratimikos solver interface of Trilinos. The goals of the three-level extension of FROSch are to provide both a flexible software design, which is compatible with the majority of the use cases of FROSch preconditioners, and an efficient parallel implementation, which scales to a large number of MPI ranks.

The object-oriented design of the FROSch based on Trilinos facilitates the recursive construction and application of the preconditioners to the coarse problem. In order to allow for a variety of solvers for the coarse and subdomain problems, FROSch uses an object-oriented interface to different solver classes, e.g., the Amesos2 package for direct linear solvers. Any solver interface in FROSch is derived from the abstract class `FROSch::Solver`, which, at construction time, only requires the system matrix and a parameter list, which defines the selected solver and its settings, as input. In the standard two-level method, the solver for the coarse problem is a sparse direct linear solver (e.g., PardisoMKL). As described in section 3, the three-level extension is obtained by simply applying the (R)GDSW preconditioner recursively to the coarse problem. Therefore, the solver for the coarse problem is chosen as an FROSch preconditioner in the parameter list. As described in [19, 18, 17] in detail, the FROSch preconditioners can, in principle, be constructed in a fully algebraic, black box way. However, the performance of the preconditioner can benefit from additional information. Therefore, in addition to certain a priori defined parameters, such as the width of the overlap and the sparse direct linear solver for the arising subproblems, we will include additional information for the construction of the next coarser level. These parameters are listed in Table 1.

First of all, a pointer to the `Xpetra::Map` describing the nonoverlapping domain decomposition on the next higher (subregion) level is stored in the parameter list; we denote this map as `Repeated Map`. It will then be used to identify the interface components on subregion level; cf. [19, 18, 17]. In subsection 5.1, we describe the procedure to build the `Repeated Map` from the nonoverlapping domain decomposition on subdomain level. In order to identify interface components and degrees of freedom (DOFs), FROSch generally orders the DOFs in a consistent way; cf. subsection 5.2. The respective ordering is passed from one to the next coarser level by providing an array of `Xpetra::Map` objects through the parameter list, where each map stores the indices for one DOF for each interface component (`DOF Maps`). Additionally, we provide a local-to-global mapping for the interface components via the `Node Map`; this is because the interface components on the coarse levels correspond to the finite element nodes on the first level; cf. Figure 1. Note that both the `Repeated Map` and the `Node Map` could, in principle, be computed from the `DOF Maps`; however, we still decided to directly pass them to the next coarser level through the parameter list. The dimension of the nullspace specifies the number of DOFs on each interface

TABLE 1
Parameter passed to the parameter list of the recursive application of FROSch.

Parameter	Description	Section
`Repeated Map`	`Xpetra::Map` for the nonoverlapping domain decomposition; interface degrees of freedom are shared by the adjacent subdomains—thus the name `Repeated Map`.	subsection 5.1
`DofsPerNode`	Number of degrees of freedom on each coarse node; equal to the dimension of the nullspace.	subsection 5.3.
`Dofs Maps`	Array of `Xpetra::Map` objects describing the distribution of the DOFs: one map corresponding to each DOF per node. The `Repeated Map` can be assembled from the `Dofs Maps`.	subsection 5.2
`Node Map`	`Xpetra::Map` corresponding to the nodes/interface components.	section 5
`Null space`	`Xpetra::MultiVector` containing a basis of the nullspace on current level as columns.	subsection 5.3

component of the coarse level. In subsection 5.3, we will describe how a basis for the nullspace on the next coarser level can be constructed algebraically based on the nullspace on the current level; it is then also stored in the parameter list as a pointer to an `Xpetra::MultiVector`.

In summary, the following steps have to be performed for the recursive application of the preconditioner.

- Partitioning of a coarse matrix to obtain the corresponding domain decomposition on the respective level.
- Definition of a consistent DOF ordering across the levels.
- Lifting of the nullspace from one level to the next coarser level.
- Establishing the communication patterns between two levels.

Based on these steps, we are able to construct FROSch preconditioners in a recursive way, obtaining three-level (R)GDSW preconditioners. These steps will be discussed in detail in the following subsections. Note that these steps will be performed algebraically only using the information provided by the previous level.

5.1. Partitioning the coarse problem. Recursive application of the FROSch preconditioner requires a nonoverlapping decomposition into subregions, which is employed to construct the overlapping subregions as well as the interface components (faces, edges, and vertices) on the subregion level; see section 3. As also mentioned in section 3, the nonoverlapping subregions are chosen as the union of nonoverlapping subdomains. Hence, in order to partition the coarse problem, we make use of the structure of the domain decomposition on the first level.

As described in [18], the nonoverlapping domain decomposition is given in form of a *repeated* Epetra or Tpetra map object, which defines the parallel distribution of the DOFs. Each subdomain is associated with a single MPI rank such that the `Repeated Map` defines the nonoverlapping subdomains. However, several OpenMP threads may be used for each subdomain. Note that for a nonoverlapping domain decomposition, the DOFs on the interface are shared by the adjacent subdomains, which also allows us to identify the interface components; cf. [19, 18] for a more detailed discussion of the identification of the interface components. Either this map object is provided as an input by the user or, as discussed in [18, 17], an algebraic reconstruction of the map can be performed in FROSch. If necessary, the reconstruction is performed in a preprocessing step before the first and second levels of the (R)GDSW FROSch preconditioner are constructed. As we will explain, the partitioning procedure on the subregion level is fully algebraic, and therefore the `Repeated Map` on the subregion level can always be constructed explicitly.

In order to partition the coarse problem, we first construct the dual graph of the domain decomposition into nonoverlapping subdomains as an `Xpetra::CrsGraph` object. This is performed based on the interface components, which are always available since they have previously been identified in the construction of the second level based on the `Repeated Map`. The dual graph contains the adjacency relations of the nonoverlapping subdomains, and we define two subdomains to be adjacent if they share an edge (in 2D) or a face (in 3D), respectively; see Figure 2 for a graphical representation in 2D. After storing all interface components as well as the index set of all subdomains which share an interface component, we can first assemble the dual graph without any additional communication. However, each MPI rank only holds the row of the corresponding subdomain, and therefore, the graph is distributed over all processes.

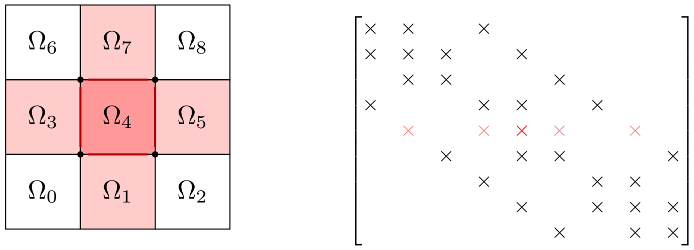

FIG. 2. *Two-dimensional structured domain decomposition into* 9 *subdomains (left) and the corresponding dual graph (right). Adjacency relations are given through the edges in* 2*D or through the faces in* 3*D, respectively.*

```
typedef Zoltan2::XpetraCrsGraphAdapter<Xpetra::CrsGraph<LO,GO,NO
    >> inputAdapter;

Teuchos::RCP<inputAdapter> adaptedGraph =
Teuchos::rcp(new inputAdapter(dualgraph,0,0));

Teuchos::RCP<Zoltan2::PartitioningProblem<inputAdapter>>problem;

problem = Teuchos::RCP<Zoltan2::PartitioningProblem<inputAdapter
    >>(new Zoltan2::PartitioningProblem<inputAdapter>(
    adaptedGraph.getRawPtr(),ZoltanParameterList.get(),
    TeuchosComm));

problem->solve();

adaptedGraph->applyPartitioningSolution(*dualGraph,
    repartionedGraph,problem->getSolution());
```

FIG. 3. *Redistribution of the dual graph using Trilinos package Zoltan*2 [37]. *The* `inputAdapter` *provides access for Zoltan*2 *to the* `Xpetra::CrsGraph` *objects such as the* `dualGraph`*, which corresponds to the dual graph to be partitioned as described in subsection* 5.1. `Zoltan2::PartitioningProblem` *sets up the model and the solution object. The call* `solve()` *runs the partitioning algorithm chosen by the user through the parameter list. Here, the MPI communicator is stored in the* `TeuchosComm` *object.*

Since there is a one-to-one relation of MPI ranks and subdomains or subregions, respectively, in FROSch, we next communicate the complete dual graph to a subset of MPI ranks, where each rank corresponds to a subregion. These MPI ranks are combined as an MPI subcommunicator, which is called `CoarseSolveComm_` in FROSch. The communication from all MPI ranks to `CoarseSolveComm_` is performed using an `Xpetra::Export` object. Then, we perform an unstructured partitioning of the dual graph using the Trilinos package Zoltan2 [37]; cf. Figure 3. Zoltan2 provides an interface to several external mesh partitioning tools, such as ParMETIS and Scotch, but also to the partitioning algorithms from the previous Zoltan package. The partition of the dual graph then results in the domain decomposition into nonoverlapping subregions.

Based on this partition, we have to repartition the coarse matrix K_0 and build a `Repeated Map` on the subregion level such that we can construct an (R)GDSW preconditioner for the coarse matrix.

Therefore, we store, in an additional `Xpetra::CrsGraph` object, the information which interface component on the subdomain level (second level) belongs to which subdomain; this information corresponds to the element list of a triangulation.

Hence, this object contains the information to construct a mapping of the DOFs of V_0 (= rows of K_0) to the subdomains. We also make use of the fact that the DOFs corresponding to the faces, edges, and vertices are always ordered in a consistent way; cf. subsection 5.2 and Figure 6. Now, in order to determine which DOFs correspond to the subregions, we communicate this graph object to the `CoarseSolveComm_` as well. Finally, using this data, we can construct the `Repeated Map` on the subregion level, which contains all necessary data about the structure of nonoverlapping domain decomposition on subregion level. Moreover, this `Repeated Map` can also be employed to construct a corresponding uniquely distributed map for the distribution of the coarse matrix.

As we have observed in [25], the choice of the partitioning algorithms can significantly influence the performance of the three-level (R)GDSW preconditioner. From our previous observations for structured meshes and domain decompositions [25], we preferred the parallel hypergraph (PHG) partitioner from Zoltan.

Here, we consider an unstructured, adaptive finite element discretization for a linear elasticity problem provided by the deal.II software library [1, 3]. The problem is defined on a square $[-1, 1]^2$ in 2D or on a cube $[-1, 1]^3$ in 3D. A constant volume force in the x-direction is applied in two circles or spheres centered at $(0.5, 0)$ and $(-0.5, 0)$ in 2D or at $(0.5, 0, 0)$ and $(-0.5, 0, 0)$ in 3D; see [1].

Here, the decomposition into subdomains is performed through the *parallel distributed triangulation* of the deal.II software library using p4est [8]. Hence, the numbering of the subdomains follows p4est's space-filling curve. We consider the grid obtained after one refinement cycle resulting in 15 986 DOFs in 2D and 44 751 DOFs in 3D. For these computation, we neglected the rotations from the nullspace. In Figure 4, the finite element triangulation and decomposition into 64 subdomains for the two-dimensional case are visualized. In Figure 5, decompositions into subregions are visualized for the different methods.

In Table 2, we report the iteration counts and dimensions of the coarse matrices K_{00} for a weak scalability study with varying numbers of subregions. For the rather small configurations investigated, the numbers of iterations are very similar.

We see that the Block method shows good results, even slightly better than our preferred method PHG. We believe that this is a result of the favorable subdomain

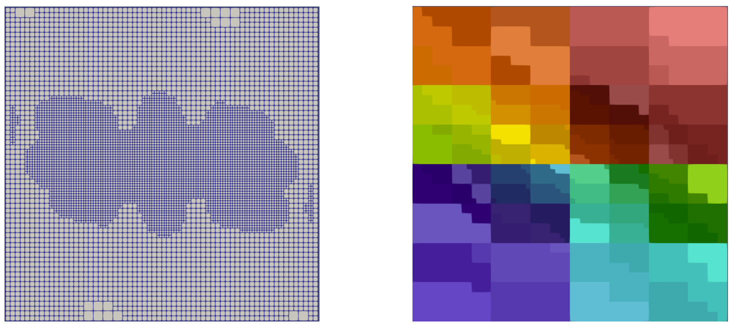

FIG. 4. *Grid for the linear elasticity benchmark problem in deal.*II *after one refinement/coarsening cycle (left) and the corresponding decomposition into* 64 *subdomains (right); each distinct color corresponds to one subdomain.*

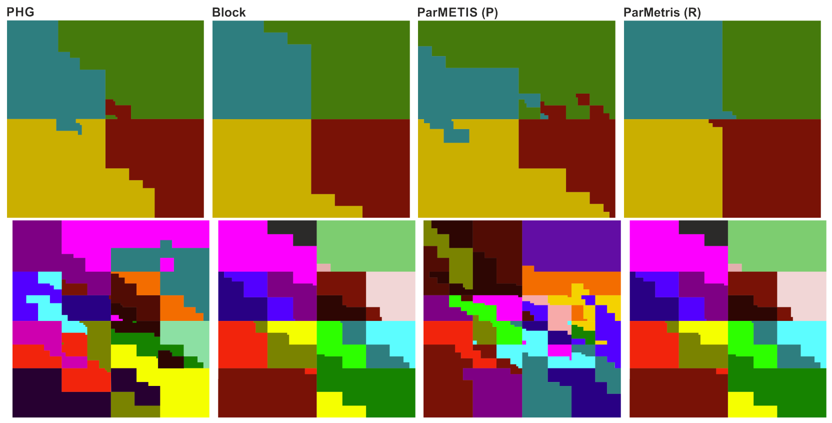

FIG. 5. *Decomposition of* 64 *subdomains of Figure* 4 *into* 4 *and* 16 *subregions using the PHG, blockwise (Block), and ParMETIS approach. ParMETIS (P) uses partitioning from scratch, and ParMETIS (R) uses repartitioning. For* 16 *subregions we obtained the same partitioning using Block and ParMETIS (R).*

TABLE 2
*Number of PCG iterations (iter) and dimension of the coarsest problem ($\dim(K_{00})$) for the three-level extension with different number of subregions and partitioning methods for the deal.*II *benchmark problem. We have $\delta = 1$ and $\Delta = 1$. In each row, the best results for the three-level GDSW and the RGDSW preconditioner are marked in* **bold**.

			Three-level GDSW preconditioner				Three-level RGDSW preconditioner			
					ParMETIS				ParMETIS	
			PHG	Block	(P)	(R)	PHG	Block	(P)	(R)
	# subr.		$\dim(K_0) = 806$				$\dim(K_0) = 240$			
2D; 64 subd. unstruct.	4	iter	54	53	**52**	**52**	**52**	**52**	54	**52**
		$\dim(K_{00})$	16	**12**	16	14	**2**	**2**	**2**	4
	15	iter	60	59	60	**56**	64	**62**	67	67
		$\dim(K_{00})$	122	**118**	286	148	**36**	38	88	58
	16	iter	**57**	58	63	58	66	**61**	67	**61**
		$\dim(K_{00})$	152	**148**	214	**148**	**48**	54	70	54
	# subr.		$\dim(K_0) = 5547$				$\dim(K_0) = 753$			
3D; 125 subd. unstruct.	4	iter	**29**	**29**	**29**	**29**	**29**	30	30	30
		$\dim(K_{00})$	57	**42**	45	45	**3**	**3**	**3**	**3**
	16	iter	**33**	**33**	**33**	**33**	**33**	**33**	**33**	**33**
		$\dim(K_{00})$	804	**504**	735	720	102	**60**	84	81
	25	iter	36	**32**	36	35	33	**32**	37	37
		$\dim(K_{00})$	1302	**576**	1614	1077	90	**51**	213	135
	32	iter	38	**37**	**37**	**37**	39	**38**	39	39
		$\dim(K_{00})$	1 677	**1281**	1 702	1 389	216	165	243	**135**

numbering provided by p4est's space-filling curve approach. As a result, the Block method performs better than ParMETIS (P), and ParMETIS (R) is not able to improve the partitioning provided by the Block method significantly (or at all).

We can conclude that for the special case of a partitioning based on space-filling curves, e.g., using the p4est library, the linear partitioning by Block is a suitable choice, even slightly better than PHG. However, as a default, since we do not use p4est throughout the rest of the paper, we continue to use PHG.

5.2. Defining a consistent ordering across levels. On each of the levels of the preconditioner, we deal with nodes and corresponding DOFs. Whereas, on the

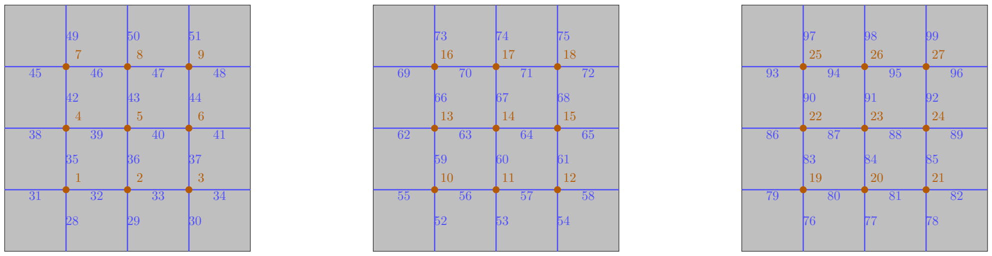

FIG. 6. *Illustration of the DOF ordering on the coarse level for a two-dimensional linear elasticity model problem with* 16 *subdomains. Each mesh shows the DOFs corresponding to one of the three rigid body modes.*

first level, the nodes just correspond to the finite element nodes, on all coarser levels they correspond to the interface components. The DOFs belonging to an interface component arise from the restriction of the nullspace to the interface components; see also subsection 5.3. In the construction of a coarse level, FROSch mostly works with the index sets of the nodes/interface components instead of the DOFs. Depending on the number of DOFs per node, this may save both computational work and communication time.

We will describe this for the GDSW case, where we have face, edge, and vertex interface components. The RGDSW case is then handled analogously. In order to allow for a consistent mapping between nodes and DOFs, FROSch uses a consistent ordering of the DOFs. In particular, FROSch first constructs individual local-to-global mappings for the vertices, edges, and faces; the mappings are, again, stored as `Xpetra::Map` objects. Then, the local-to-global mapping of all DOFs is assembled from the individual maps by appending them in a certain way. In particular, if we have k DOFs per interface component, the global map is assembled by appending the vertex map k times, followed by k edge maps and k face maps. The resulting index ordering is illustrated in Figure 6 for a two-dimensional linear elasticity model problem with 16 subdomains. Here, the nullspace is three-dimensional, corresponding to two translations and one linearized rotation, and therefore, we generally have three DOFs per node on each coarse level. Of course, the restriction of the nullspace to certain interface components may result in linear dependencies. As discussed in subsection 5.5, we deal with these linear dependencies in a way which does not remove any DOFs from our ordering.

5.3. Lifting the nullspace to the next coarser level. The construction of the coarse basis functions is based on the nullspace of the global Neumann matrix; see section 2. The nullspace can be provided as an input to FROSch. For our model problem (4.1), the nullspace is spanned by the (linearized) rigid body modes. In 3D, the space is spanned by three translations

$$r_1 := \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad r_2 := \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad r_3 := \begin{bmatrix}0\\0\\1\end{bmatrix} \tag{5.1}$$

and three linearized rotations

$$r_4 := \begin{bmatrix}x_2-\hat{x}_2\\-(x_1-\hat{x}_1)\\0\end{bmatrix}, \quad r_5 := \begin{bmatrix}-(x_3-\hat{x}_3)\\0\\x_1-\hat{x}_1\end{bmatrix}, \quad r_6 := \begin{bmatrix}0\\x_3-\hat{x}_3\\-(x_2-\hat{x}_2)\end{bmatrix}, \tag{5.2}$$

where $\hat{x}$ is the origin of the rotation. To build the FROSch preconditioner recursively for the next coarser level, we need to build a basis of the nullspace of the current level. In particular, as is well known in multilevel methods, it is crucial for numerical scalability of the preconditioner that the coarse level can represent the nullspace; see, e.g., [34].

The main difficulty in the construction concerns the rotations, as their computation requires geometric information. We would, however, like to avoid the use of geometric information on the coarser levels. Instead, we will make use of the fact that the basis of the nullspace of the coarse matrix can be given explicitly. We will use Theorem 5.1 and show that its assumptions are generally satisfied for (R)GDSW coarse spaces. Here, we use that the columns of the matrix Φ in the (R)GDSW preconditioner are linear independent, i.e., a basis of the coarse space. Furthermore, by construction, the coarse basis functions form a partition of the nullspace; this follows because they are restrictions of the nullspace to the interface components which are extended in an energy minimizing way to the interior.

For our linear elastic model problem (4.1), the global Neumann matrix K_N and and the coarse matrix $K_0 = \Phi^T K \Phi$ are symmetric positive semidefinite.

Theorem 5.1. *Let K_N be the global symmetric positive semidefinite Neumann matrix, and let the columns $z^1, \ldots, z^k$ of Z span its k-dimensional nullspace. Furthermore, let*

$$\Phi = \begin{pmatrix} \varphi_1^1 & \cdots & \varphi_{n_1}^1 & \quad\cdots\quad & \varphi_1^k & \cdots & \varphi_{n_k}^k \end{pmatrix}$$

have linear independent columns, where the φ_i^j are the basis functions of the coarse space V_0, and let

$$z^j = \sum_{i=1}^{n_j} \varphi_i^j, \quad j = 1, \ldots, k. \tag{5.3}$$

Then, the vectors $z_0^1, \ldots, z_0^k \in \mathbb{R}^m$ form a basis of the nullspace Z_0 of the symmetric positive semidefinite coarse matrix $K_0 = \Phi^T K_N \Phi \in \mathbb{R}^{m \times m}$, where $m = kn$ and $n := \sum_{k=1}^{j} n_k$. The vectors are of the form

$$\begin{aligned} (z_0^j)_i = & \quad \begin{cases} 1 & \text{if } \sum_{k=1}^{j-1} n_k < i \leq \sum_{k=1}^{j} n_k, \\ 0 & \text{else} \end{cases} \\ \text{or, equivalently,} \quad z_0^j = & \quad (0 \ \cdots \ 0 \quad \underbrace{1 \ \cdots \ 1}_{j\text{th block (length } n_j)} \quad 0 \cdots 0)^T. \end{aligned} \tag{5.4}$$

Proof. First, let $0 \neq v \in \mathbb{R}^m$. Then, v is in the nullspace of $K_0 \neq 0$ if and only if

$$0 = v^T K_0 v = v^T \Phi^T K_N \Phi \, v.$$

Since the columns of Φ are linear independent, we have $0 \neq \Phi \, v =: w$ and $w^T K_N w = 0$, which is the case if and only if w is in the nullspace of K_N. This means that

$$w = \sum_{j=1}^{k} c_j z^j.$$

Now, assuming (5.4), we have

$$\sum_{i=1}^{n_j} \varphi_i^j = \begin{pmatrix} \varphi_1^1 & \cdots & \varphi_{n_1}^1 & \quad \cdots \quad & \varphi_1^k & \cdots & \varphi_{n_k}^k \end{pmatrix} z_0^j = \Phi\, z_0^j,$$

and thus,

$$w = \sum_{j=1}^{k} c_j z^j \overset{(5.3)}{=} \sum_{j=1}^{k} c_j \sum_{i=1}^{n_j} \varphi_i^j = \sum_{j=1}^{k} c_j \Phi\, z_0^j = \Phi \sum_{j=1}^{k} c_j z_0^j.$$

In total, we have

$$\Phi\, v = w = \Phi \sum_{j=1}^{k} c_j z_0^j$$

which, since the columns of Φ are linear independent, is equivalent to

$$v = \sum_{j=1}^{k} c_j z_0^j,$$

i.e., any vector v in the nullspace of K_0 can be written as a linear combination of the vectors z_0^j, $j = 1, \ldots, k$, containing only blocks of zeros and ones, as given in (5.4). ☐

In order to apply (5.1) to our (R)GDSW coarse spaces, we have to check whether the assumptions are satisfied.

LEMMA 5.2. *For the (R)GDSW-type coarse spaces, we have that the columns of* Φ *are linear independent and, after possibly reordering the columns, they satisfy* (5.3).

Proof. The coarse basis functions of the (R)GDSW coarse spaces are obtained by computing the energy minimizing extension of the interface functions Φ_Γ. The columns of Φ_Γ are the restrictions of the nullspace Z to the interface Γ. Hence, after reordering the columns, we have $\Phi_\Gamma = \begin{pmatrix} \phi_1^1 & \cdots & \phi_{n_1}^1 & \quad \cdots \quad & \phi_1^k & \cdots & \phi_{n_k}^k \end{pmatrix}$, and

$$z^j|_\Gamma = \sum_{i=1}^{n_j} \phi_i^j, \quad j = 1, \ldots, k. \tag{5.5}$$

The energy minimizing extension $\mathcal{H}(z^j|_\Gamma)$ of $z^j|_\Gamma$ into the interior is the function which coincides with $z^j|_\Gamma$ on the interface Γ and has minimum energy. This is just the function z^j itself because z^j is in the nullspace and therefore has the minimum energy of 0. Therefore,

$$z^j = \mathcal{H}(z^j|_\Gamma) = \sum_{i=1}^{n_j} \mathcal{H}(\phi_i^j), \quad j = 1, \ldots, k.$$

By construction, the basis functions φ_i^j, that is, the columns of Φ, are the energy minimizing extension of the ϕ_i^j, and hence,

☐

$$z^j = \sum_{i=1}^{n_j} \varphi_i^j, \quad j = 1, \ldots, k.$$

Finally, combining (5.1) and (5.2), we obtain the following Corollary 5.3.

Corollary 5.3. *A basis of the nullspace of the (R)GDSW coarse matrix is given by* (5.4)*. This implies that no explicit geometric information is needed on the third level, i.e., also the rotations can be interpolated by vectors using ones and zeros.*

Following Corollary 5.3 we build the nullspace for the coarse level explicitly. This construction can also be used for a multilevel extension to the FROSch framework.

Note that in the global matrix approach [6, 5] in algebraic multigrid for elasticity the Global Matrix (GM) variant 1 also does not need to interpolate rotations after the first level. BoomerAMG has scaled to 262 144 cores for three-dimensional linear elasticity using the global matrix approach [5].

5.4. Establishing the communication pattern between levels. As described in subsection 5.1, the coarse problem is distributed over a subset of MPI ranks on the subcommunicator `CoarseSolveComm_`. We have to redistribute the coarse matrix in the construction and iteration vectors in the Krylov iteration correspondingly. We have observed that, if the difference in size of the communicators is large, the redistribution, which is performed using `Xpetra::Export`, can be expensive; see, for example, the discussion in [19]. It is likely that this is due to the current implementation of the communication pattern of the import and export functions in Epetra and Tpetra; as can also be seen in Figure 7, the communication essentially corresponds to `MPI_Send` and `MPI_Irecv` calls. In order to speed up this communication, it is split into several steps. According to the number of steps to gather (and scatter) the matrix and vectors, which we denote as `Gathering Steps`, we recursively set the number of processes in each step, where we start with all MPI processes:

```
numProcsGatheringStep = int(numProcsGatheringStep/gatheringFactor);
```

Here, *gatheringFactor* is the reduction factor for the number of processes. It is computed as

```
size = MpiComm_->getSize();
gatheringFactor = pow(size/numProcsCoarse,1/GatheringSteps).
```

The number of processes is reduced until the predefined size of `CoarseSolveComm_` is reached; see also Figure 8. This results in a number of `Gathering Steps`, where, in each step, the rows are distributed evenly. First, an `Xpetra::Map` and an `Xpetra::Export` object for each step is constructed, and then the communication can be performed in both ways using either the `doExport` function or the `doImport` function;

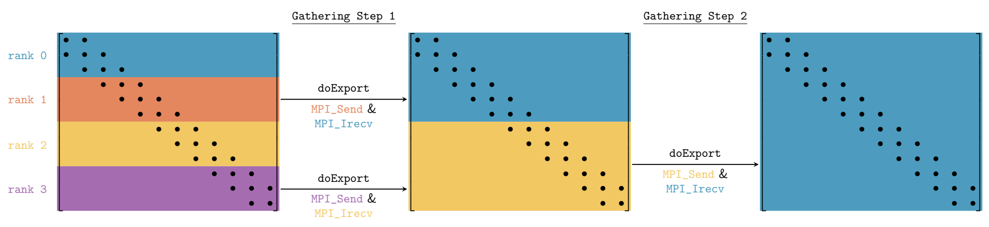

Fig. 7. *Schematic representation of two* `Gathering Steps` *applied to a parallel distributed matrix for the case of four MPI ranks: First, the matrix is distributed over four ranks (blue, red, yellow, and purple). Then, by using* `doExport` *we reduce the number of ranks by a factor of two in each step. After two* `Gathering Steps`*, the complete matrix has been communicated to rank* 0 *(blue). Internally, performing* `doExport` *corresponds to calls of* `MPI_Send` *and* `MPI_Irecv` *on the sending and receiving ranks, respectively.*

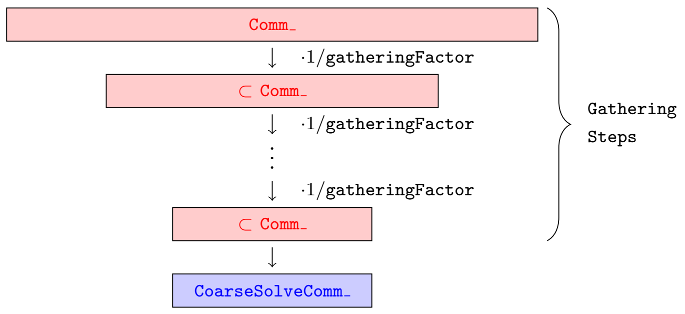

FIG. 8. *Reducing the number of MPI ranks from the global communicator* `Comm_` *(e.g.,* `MPI_COMM_WORLD`*) to* `CoarseSolveComm_`*. All intermediate steps are performed on subsets of the original communicator without creating intermediate subcommunicators. In the last step, which is not part of the* `Gathering Steps`*, no communication is necessary since the data is already available on the MPI ranks of* `CoarseSolveComm_`*.*

```
RCP<SerialQRDenseSolver<LO,SC> > qRSolver(new
    SerialQRDenseSolver<LO,SC>());
qRSolver->setMatrix(PhiTPhi);
qRSolver->factor();
qRSolver->formQ();
qRSolver->formR();
```

FIG. 9. *Forming the QR-factorization using* `Teuchos::SerialQRDenseSolver`*.*

in Figure 7, we show a schematic representation of two `Gathering Steps` applied to a matrix.

5.5. Detecting linear dependencies. As discussed in subsection 5.3, for our three-dimensional linear elasticity model problem (4.1), the nullspace is spanned by three translation and three linearized rotations. However, if the construction of the coarse space is performed as described in section 2, the columns of the matrix Φ may be linear dependent. For example, the restriction of the nullspace to a vertex yields only a three-dimensional space.

In order to make sure that the coarse matrices K_0 and, for the three-level method, K_{00} are invertible, we remove the linear dependent column vectors from Φ. This is performed before building the coarse matrix from the Galerkin product. In our implementation, we perform a QR-factorization of $\Phi^T\Phi$ using the `Teuchos::SerialQRDenseSolver`, which is written on top of BLAS and LAPACK; see Figure 9 for the respective code snippet. We use $\Phi^T\Phi$ instead of $K_0 = \Phi^T A\Phi$ to reduce computation cost; as a consequence, we save one parallel sparse matrix-matrix multiplication.

If a diagonal entry of the matrix R is smaller than a threshold, the vector is considered linear dependent. We then replace the vector in Φ by a null vector, i.e., a vector containing only zero entries. This yields zero rows and columns in K_0 for the linear dependent vectors. In order make K_0 invertible, we replace those rows and columns by the suitable unit vectors. As a result, the size of the coarse matrix is always equal to the number of interface components times the dimension of the nullspace. Hence, we always maintain the DOF ordering described in subsection 5.2.

6. Parallel results. In this section, we focus on the weak parallel scalability of the three-level extension of the FROSch framework. We perform tests on the ALCF

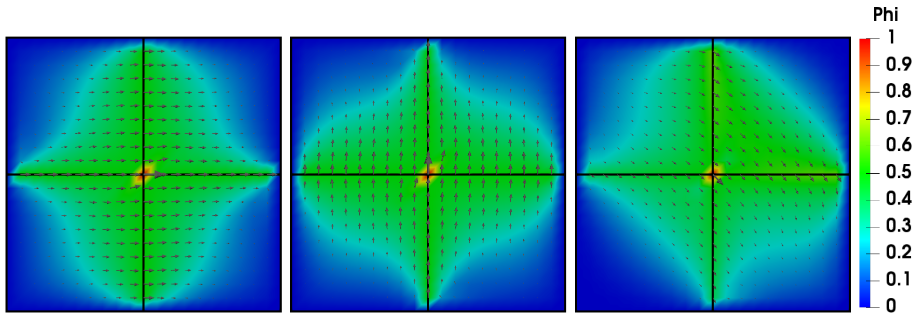

FIG. 10. *The three RGDSW coarse basis functions of type* Option 1 *for linear elasticity corresponding to the node in the middle for a structured domain decomposition in 2D: translation in x direction (left), translation in y direction (middle), and linearized rotation (right). The plot only shows the four subdomains where the functions are nonzero.*

supercomputer Theta and compare to previous results reported for the SuperMUC-NG supercomputer in [22]. The Intel Compiler 19.1.0.166 with Cray PE version 2.6.5 is used. The local subdomain and subregion problems and the coarse problem on the third level are solved using PardisoMKL [7] using one or several OpenMP threads. We use the PCG method as the Krylov iteration method, employing *Pseudo Block CG* from the Trilinos package Belos; in our case, *Pseudo Block CG* reduces to standard CG. We use a relative stopping criterion $\|r_k\|/\|r_0\| < 10^{-6}$, where r_0 is the initial residual and r_k the residual after the kth iteration step. To determine the interface, we use a map based on the structured decomposition into nonoverlapping subdomains on the first level (*Geometric Map*).

We always use the (algebraic) RGDSW coarse space denoted as Option 1 in [13, 23]. Given a basis of the nullspace, the respective basis functions can be constructed algebraically; cf. Figure 10 for a visualization of the corresponding coarse basis functions for the two-dimensional case.

We first present new computational results for FROSch obtained on the ALCF Theta supercomputer (Intel Xeon Phi 7230 64C 1.3GHz, Aries interconnect; 64 cores per node), which is currently ranked 39th in the TOP500 list. As we observe unexpected performance problems on Theta in the pure MPI setup for large numbers of cores, for comparison, we also discuss results obtained earlier [22] on the SuperMUC-NG supercomputer (Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path; 48 cores per node) of the Leibniz Supercomputing Centre (LRZ) in Garching, Germany, which is currently ranked 15th in the TOP500 list. For our comparison, we will revisit fine time timers on SuperMUC-NG not considered in [22].

In the following subsections, we will discuss timers on different levels of detail. We denote the total time to solution, including the finite element assembly of the problem, by *Total Time*, the time to build the FROSch preconditioner by *Setup Time*, and the time to perform the Krylov iteration by *Krylov Time*. The sum of the *Setup Time* and the *Krylov Time* is the *Solver Time*; see also Figure 11.

The *Setup Time* consists of two phases. The first phase is the initialization (*Init. Time*) of the preconditioner. For the first two levels of the preconditioner, this phase deals with the part of the setup which depends on the structure of system matrix K but is independent of the values of the matrix entries. Most importantly, it contains the computation of the overlapping subdomains, the identification of the interface components, and the computation of interface values of the coarse basis functions, Φ_Γ. Moreover, for setting up the third level, it also includes the computation of

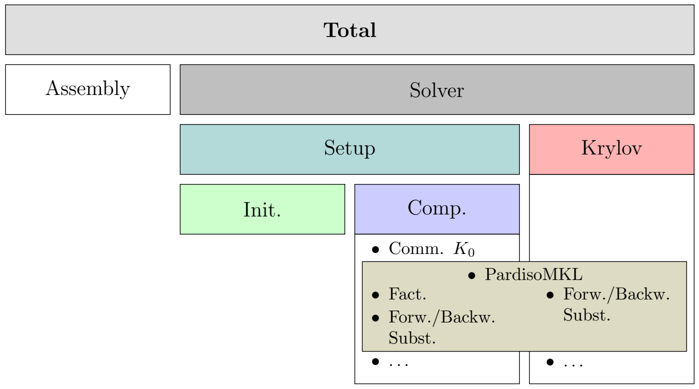

FIG. 11. *Illustration of the coarse and fine grained timers used in the different tables. The above positioned timer includes the ones below. The PardisoMKL Time is partly included in the Comp. Time and the Krylov Time. For a detailed description see subsection* 6.1.

the dual graph, the partitioning of the coarse problem, and the computation of the `Repeated Map`. All remaining parts of the setup of the third level are only performed after the computation of the coarse matrix itself, which is part of the second phase, the compute phase (*Comp. Time*). It includes the initialization phase on subregion level as well as, on each level, the computation and factorization of the overlapping matrices, the computation of the energy minimizing extensions, and the computation and communication of the coarse matrix to the coarse communicator. Here, we will specifically report the time for communicating the matrix K_0 (*Comm.* K_0 *Time*).

We will also discuss the time spent on the sparse direct solver, Pardiso MKL (*PardisoMKL Time*). This is split into the factorization (*Fact. Time*) and forward/backward substitution (*Forw./Backw. Subst. Time*). There are three different occurrences of direct solvers: the solution of the local problems on the overlapping subdomains/-subregions, the computation of the energy minimizing extensions on subdomain and subregion level, and the solution of the coarse problem K_{00}. All matrix factorizations are computed within the compute phase. Moreover, most forward/backward substitutions are performed during the Krylov iteration. The only exception is the computation of the interior values of the coarse basis functions Φ_I by energy minimizing extensions, which is necessary for the computation of the coarse matrices and therefore fully contained within the compute phase.

For more details on the implementation of the different operations, see [19, 18] and section 5, and for a graphical illustration of the correlations of the timers, see Figure 11.

6.1. Results for pure MPI on Theta. For our first scaling tests, we use 64 MPI ranks on each Intel Knights Landing (KNL) node with 64 cores. Our model problem (4.1) is applied such that each process owns 15^3 nodes. For the subdomain overlap, we use an algebraic overlap of one layer of elements. On the subregion level, we instead use zero layers of subdomains of overlap starting from the nonoverlapping domain decomposition, i.e., only the interface Γ is shared by different processor cores. The decomposition into subregions is performed using the PHG partitioning algorithm from Zoltan, resulting in an unstructured decomposition; cf. subsection 5.1. However, the number of subregions is chosen such that each subregion consists of approximately 8 subdomains.

In Table 3, we present the weak scalability results using 64 ranks per node on Theta. Good numerical scalability, in terms of Krylov iterations, is obtained: the baseline for the iteration counts of three-level GDSW methods is typically larger compared to the two-level method due to the inexact coarse solve. Moreover, a slight increase in the number of iterations for larger number of subdomains is expected since for unstructured decompositions the asymptotic behavior is typically reached later. Indeed, for larger numbers of subdomains (>100 000), the iteration counts are almost constant.

With respect to the parallel scalability, already for 175 616 cores, the *Total Time* has increased to 291 seconds; see Table 3. A computation with more than 200 000 cores exceeded the walltime and is not reported here. This parallel scalability problem is unexpected, since good scalability of of the three-level GDSW preconditioner implementation has been achieved previously [20, 21].

The problem seems to be in the construction of the preconditioner: For 110 512 cores, the *Setup Time* takes almost 80% of the computing time (see Table 3 and Figure 12), which is clearly not acceptable.

To determine the cause of the scalability problems, we consider more fine grained timings, presented in Table 4. In Table 4, we see that the main issue is the communication of the coarse matrix by the gathering procedure (*Comm.* K_0 *Time*). Note that once this coarse communication pattern is established in the setup as described in subsection 5.4, in every Krylov iteration the corresponding `Gathering Steps` have to be performed for the iteration vectors. Therefore, since the coarse communication occurs in every iteration, also the *Krylov Time* suffers from the same problem as the setup, which is visible in Table 3, where the *Krylov Time* increases to 370.39s. We have observed problems with this operation earlier; however, on other supercomput-

TABLE 3
Weak parallel scalability results on Theta with 64 *MPI rank on each node with* 64 *cores. By Iter we denote the number of Krylov iterations. The computation using* 512 *nodes failed for unknown reasons and could not be repeated. The scalability problem is marked in* **bold**. *The sum of the Setup Time and the Krylov Time is the Solver Time; see also Figure* 11.

#Nodes	#Cores	#Subd.	#Subr.	$\kappa(M^{-1}K)$	Iter	Total Time	Solver Time	Setup Time	Krylov Time
64 MPI ranks per Theta node									
27	1 728	1 728	4	66.94	69	23.85 s	19.61 s	12.60 s	7.11 s
125	8 000	8 000	16	87.56	80	28.71 s	24.27 s	16.87 s	7.40 s
216	13 824	13 824	27	87.10	83	35.88 s	31.02 s	21.84 s	9.18 s
512	32 768	32 768	64						
1 728	110 512	110 512	216	101.28	93	**94.83 s**	**86.07 s**	**73.88 s**	**12.19 s**
2 744	175 616	175 616	343	97.29	93	**291.21 s**	**283.58 s**	**116.01 s**	**167.57 s**
3 049	195 512	195 512	381	97.50	92	**553.35 s**	**545.66 s**	**175.27 s**	**370.39 s**

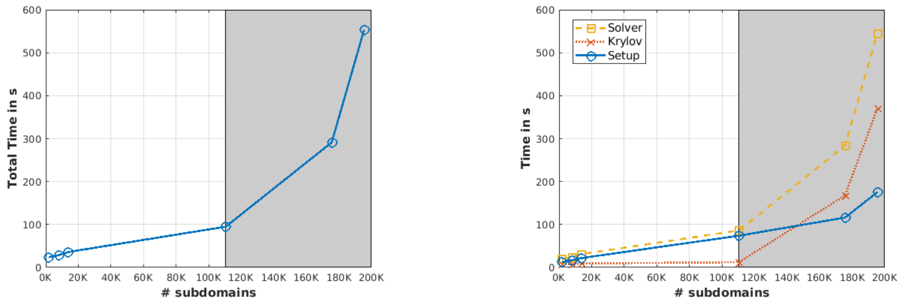

FIG. 12. *Weak parallel scalability with* 64 *MPI ranks per node. By Solver Time we denote the sum of Setup Time and Krylov Time, which is the total time of the FROSch preconditioner; see Table* 3 *for the data. The grey background indicates the loss of scalability.*

Table 4

Detailed subtimers for the Setup Time on Theta for 64 *MPI ranks; see Table* 3*. The scalability problem is marked in* **bold***. Comm.* K_0 *Time and Fact. Time are part of the Comp. Time; the sum of Init. Time and Comp. Time is the Setup Time; also see Figure* 11*.*

#Nodes	#Cores	#Subd.	#Subr.	Size K_0	Size K_{00}	Init. Time	Comp. Time	Comm. K_0 Time	Fact. Time
64 MPI ranks per Theta node									
27	1 728	1 728	4	7 986	6	2.01 s	10.59 s	0.61 s	4.51 s
125	8 000	8 000	16	41 154	156	2.18 s	14.69 s	2.15 s	4.93 s
216	13 824	13 824	27	73 002	360	2.61 s	19.22 s	4.64 s	7.79 s
512	32 768	32 768	64						
1728	110 512	110 512	216	622 938	5106	9.06 s	**64.83 s**	**40.74 s**	10.32 s
2 744	175 616	175 616	343	998 250	8 622	9.05 s	**106.96 s**	**77.47 s**	10.66 s
3 049	195 512	195 512	381	1 111 158	9 852	10.56 s	**164.70 s**	**132.58 s**	12.01 s

Table 5

Results for different numbers of `Gathering Steps` *for the case of* 13 824 *cores using* 64 *MPI ranks on each node. Increasing the number of steps does not improve the timings.*

# Gathering Steps	Total Time	Comm. K_0 Time	Krylov Time
3	35.88 s	4.64 s	9.18 s
4	38.90 s	6.21 s	8.66 s
5	42.01 s	7.34 s	10.51 s

Table 6

Weak parallel scalability results on Theta with 32 *MPI rank on each node with* 64 *cores. By Iter we denote the number of Krylov iterations. The scalability problem is marked in* **bold***.*

#Nodes	#Cores	#Subd.	#Subr.	$\kappa(M^{-1}K)$	Iter	Total Time	Solver Time	Setup Time	Krylov Time
32 MPI ranks per Theta node									
54	3456	1728	4	66.94	69	19.52 s	16.98 s	12.06 s	4.92 s
250	16000	8000	16	87.56	80	25.24 s	22.64 s	16.10 s	6.54 s
432	27648	13824	27	87.10	83	31.28 s	28.34 s	21.52 s	6.82 s
1024	65536	32768	64	83.18	82	49.80 s	**46.23 s**	**33.95 s**	12.28 s
3456	221184	110512	216	101.28	93	86.62 s	**79.24 s**	**69.81 s**	9.43 s

Table 7

Detailed subtimers for the Setup Time on Theta for 32 *MPI ranks per node.*
The sum of Init. Time and Comp. Time is the Setup Time; Comm. K_0 *Time and Fact. Time are part of the Comp. Time. The scalability problem is marked in* **bold***.*

#Nodes	#Cores	#Subd.	#Subr.	Size K_0	Size K_{00}	Init. Time	Comp. Time	Comm. K_0 Time	Fact. Time
32 MPI ranks per Theta node									
54.	3 456	1 728	4	7 986	6	1.55 s	10.44 s	0.62 s	4.47 s
250	16 000	8 000	16	41 154	156	2.09 s	14.36 s	2.11 s	4.86 s
432	27 648	13 824	27	73 002	360	2.42 s	19.45 s	5.12 s	5.09 s
1024	65 536	32 768	64	97 200	300	3.21 s	**30.83 s**	**14.01 s**	5.42 s
3456	221 184	110 512	216	622 938	5106	8.45 s	**61.05 s**	**38.25 s**	7.35 s

ers, increasing the number of hierarchical `Gathering Steps` (see subsection 5.4) has solved this problem. Here, this is not successful; see Table 5.

Assuming that the problem may be caused by MPI congestion, we have also considered a setup using only 32 MPI ranks per node with 64 cores, i.e., half of the cores are idle in this setup. The results in Table 6 and Table 7 are, however, not very different from results with 64 MPI ranks per node in Table 3 and Table 4: we see a steep increase in the *Setup Time* which is almost identical to the case with 64 MPI ranks per node. In all cases, this increase results from the *Comm.* K_0 *Time*.

For comparison, we now consider the scalability of this operation on SuperMUC-NG, where we use 48 MPI ranks per node.

6.2. SuperMUC-NG using 48 MPI ranks per node. We have performed computations on the SuperMUC-NG supercomputer [22], where the FROSch preconditioner showed a significantly better weak parallel scalability than achieved in subsection 6.1. For comparison, we here report the results for a subset of the computations from [22], using the *Geometric Map* on the first level, which is consistent with the setup used here for Theta. We revisit the fine grained timers, which were not reported in [22], to perform a comparison with the results on Theta.

Let us first briefly discuss the results obtained on SuperMUC-NG. In Table 8, we report the *Solver Time*, which is the sum of the *Setup Time* and the *Krylov Time*. On SuperMUC-NG, we choose the problem size such that each process owns 20^3 nodes and one layer of the subdomains for the subregion overlap Δ. Otherwise the setup corresponds the one used for Theta. Despite the larger problem size, the computation on SuperMUC-NG is faster than on Theta, e.g., we have 40.8s for 85 184 cores on SuperMUC-NG for the *Solver Time* which compares to 86.07s for 110 512 cores on Theta. This is partly a result of the faster processor cores of SuperMUC-NG (Xeon Platinum 8174 24C 3.1GHz), which have a significantly better floating point performance compared to Theta's low energy cores (Intel Xeon Phi 7230 64C 1.3GHz). For completeness, we report results for the two-level and the three-level methods in Table 8. We see that, compared to the two-level method, a higher number of iterations is reported for the three-level method; the three-level method is, however, faster than the two-level method if 27 000 or more subdomains are used.

The other main contribution to the faster *Solver Time* is the *Comm.* K_0 *Time*. In particular, we see that on SuperMUC-NG the communication for the coarse operator, which also uses 3 `Gathering Steps`, is at the order of one second for all runs; see Table 10 and Figure 13. This is in striking contrast to the results on Theta, where the communication concerning the coarse operator clearly dominates the *Solver Time* for a large number of cores. On Theta, already for 32 768 MPI ranks, the *Comm.* K_0 *Time* is 14.01 seconds; see Table 7. For 110 512 MPI ranks, it is 38.25s in Table 7 and 40.74s in Table 3. For larger number of ranks, the coarse communication time on Theta is prohibitive for pure MPI with our current implementation.

We have discussed our coarse communication pattern and our measurements with ALCF (Theta) and LRZ (SuperMUC-NG). Theta uses a three-level dragonfly

TABLE 8
Weak scalability results for the Solver Time obtained on SuperMUC-NG; subset of the data in [22, Table 1].

			Two-level			Three-level		
#Nodes	#Cores = # Subd.	#Subr.	$\kappa(M^{-1}K)$	Iter	Solver Time	$\kappa(M^{-1}K)$	Iter	Solver Time
48 MPI ranks per SuperMUC-NG node								
21	1 000	4	51.45	57	15.11 s	90.46	72	16.99 s
288	13 824	27	53.61	61	38.40 s	116.19	90	24.89 s
563	27 000	64	53.77	62	87.28 s	122.18	95	30.87 s
819	39 304	125	53.82	62	153.88 s	128.39	98	35.12 s
1 334	64 000	216	-	-	-	135.58	98	37.29 s
1 775	85 184	275	-	-	-	108.49	99	40.80 s

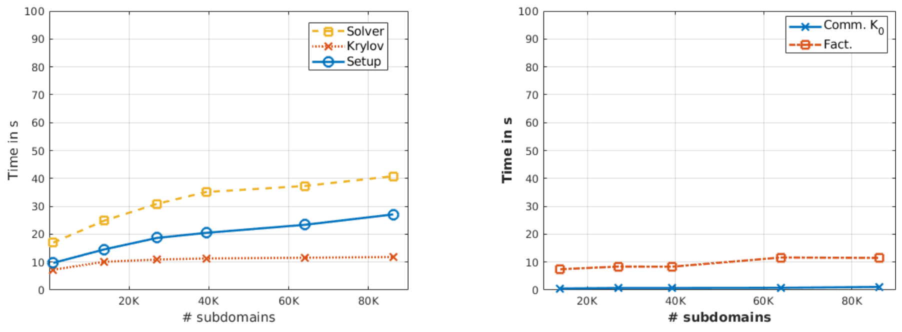

FIG. 13. *Weak parallel scalability for the Solver Time, Krylov Time, and Setup Time (right) obtained on SuperMUC-NG; see Table* 8 *for the data. Time to communicate the coarse matrix (Comm. K_0) and time for the factorization (Fact.) (right); see Table* 10 *for the data.*

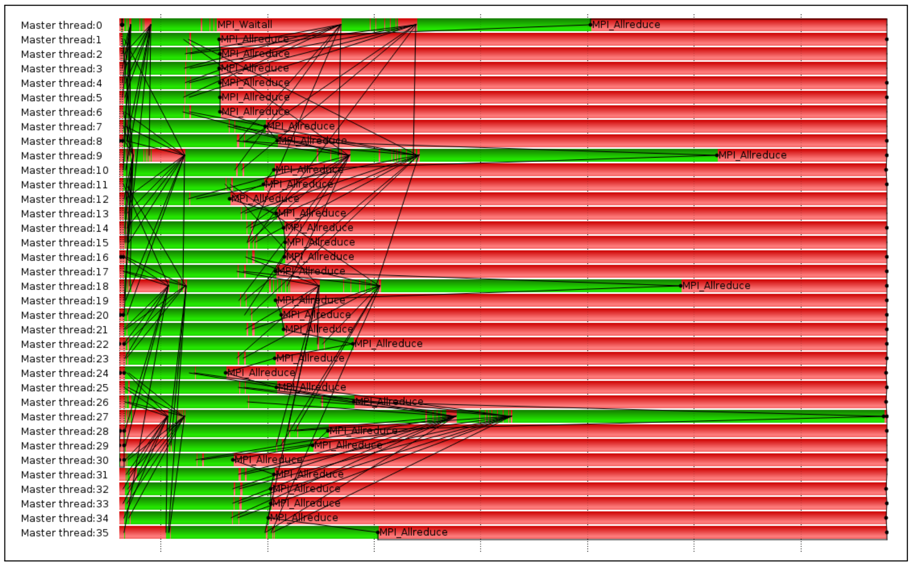

FIG. 14. *MPI-trace of the communication of the coarse matrix to the coarse communicator for* 36 *cores. We apply* 4 *processes on the coarse communicator. The test was performed on the high performance computing cluster of the Technische Universität Bergakademie Freiberg using the Vampir tracing tool* [15].

interconnect, where on the first level 4 nodes are connected to a single Aries router. The second level is a full mesh connecting the 96 routers. The 96 routers, corresponding to two racks, form a single group. On the global level, each group of 384 nodes is connected by active optical links to each other group; however, the bandwidth of the global links between groups is limited. It is plausible that our coarse communication pattern, which is not adapted to the network architecture, is problematic for Theta's dragonfly Aries interconnect and can result in global link congestion.

Our communication pattern, which basically consists of many-to-few communication using `MPI_Send/MPI_Irecv` (see Figure 14) for an example) may result in worst case traffic, i.e., many nodes of one group may send data to a single node of a different group; see also adversarial traffic patterns for dragonfly interconnects [14]. It is also known that Theta's dragonfly latency across groups, especially under interconnect noise, due to the sharing of the interconnect across several running jobs, can be worse than expected on Theta [10]. On the other hand SuperMUC-NG's superior fat tree interconnect seems to handle our coarse communication well.

To address the problem on dragonfly-type interconnects, the coarse communication pattern would need to be adapted to the network architecture: the `Gathering Steps` should first aggregate the data within groups and then aggregate the data of the different groups.

We will also consider reimplementing the coarse communication using different MPI primitives.

6.3. Hybrid parallelization with MPI and OpenMP. As an alternative to pure MPI, we also consider hybrid MPI/OpenMP parallelization. Here, we can benefit from the shared memory parallelization in the Tpetra linear algebra through Kokkos [9] as well as in the PardisoMKL sparse direct solver.

In order to find the most efficient use of OpenMP parallelization on the node, we perform test runs on 216 nodes with 13 824 cores using pure MPI as well as 2, 4, and 8 threads per MPI rank. For these tests, each process owns 20^3 finite element nodes, which results in a total problem size of 41.472×10^6. Hence, in these tests, the same local problem size is used as on SuperMUC-NG in subsection 6.2. The results are shown in Table 9. In all cases, we have 74 Krylov iterations. We see that using 2 threads reduces the *Total Time* from 39.03s to 30.51s. Using 4 threads results in 36.21s which is slower than when using 2 threads. Using 8 threads is the fastest option in terms of *Total Time* (28.01 seconds); however, the advantage is minor compared to using only 2 threads. The slight reduction in *Total Time* is achieved, although PardisoMKL performs worse than when using a single thread. The (very small) performance gain therefore must be a result of the underlying parallel linear algebra kernels from Kokkos kernels used by Tpetra.

After these results, we now try to scale up a hybrid MPI/OpenMP setup, using 2 threads per MPI rank, to almost the complete Theta supercomputer.

TABLE 9
Results for the case of 13 824 *cores (*216 *nodes) using 8 MPI processes on each node of the Theta supercomputer. We have* 1 728 *subdomains and* 4 *subregions. Each process owns* 20^3 *nodes. Different numbers of OpenMP threads are compared. Amesos*2 *uses PardisoMKL to factorize the linear system.*

#OpenMP Threads	Total Time	Forw./Backw. Subst. Time	Fact. Time	PardisoMKL Time
8 MPI ranks per Theta node				
1	39.03	8.12 s	13.12 s	21.24 s
2	30.51	4.43 s	9.65 s	14.08 s
4	36.21	5.48 s	8.50 s	13.98 s
8	28.01	8.25 s	14.68 s	22.93 s

TABLE 10
Detailed subtimers for the Setup Time on SuperMUC-NG; see Table 8*. Init. + Comp. = Setup. We see that the Comm.* K_0 *Time is at the order of a second for all runs on SuperMUC-NG.*

#Nodes	#Cores	#Subd.	#Subr.	Size K_0	Size K_{00}	Init. Time	Comp. Time	Comm. K_0 Time	Fact. Time
48 MPI ranks per SuperMUC-NG node									
288	13 824	13 824	27	73 002	366	1.68 s	12.70 s	0.53 s	7.43 s
563	27 000	27 000	64	215 622	1 056	3.55 s	15.11 s	0.72 s	8.39 s
819	39 304	39 304	125	355 914	2 508	5.55 s	14.93 s	0.71 s	8.36 s
1 334	64 000	64 000	216	355 914	4 980	7.29 s	16.97 s	0.78 s	11.65 s
1 775	85 184	85 184	275	477 042	6 432	9.57 s	17.52 s	1.08 s	11.51 s

6.4. Results for up to 220 000 cores for hybrid MPI/OpenMP. The setup of these computations is identical to the one describe in subsection 6.1, i.e., we use 15^3 nodes for each process.

We are able to scale from 3 456 cores (54 nodes) to up to 221 184 cores (3 456 nodes), which is almost the complete Theta machine (4 392 nodes). The *Total Time* increases from 16.96 s for 3 456 cores up to 78.96 s for 221 184 cores; see Table 11. As for the other test cases the *Setup Time* on Theta is the most expensive part of our computation once we go beyond 32 768 MPI ranks. While the initialization of our preconditioner scales well, the *Comp. Time* again suffers from the coarse communication; see Table 12 and Figure 15.

However, we see that, using a hybrid MPI/OpenMP setup, FROSch can make use of almost the complete Theta supercomputer with an acceptable efficiency. The

TABLE 11

Weak scalability on Theta using hybrid MPI/OpenMP. Two OpenMP threads per rank. One OpenMP thread per core. By Solver Time we denote the sum of Setup Time and Krylov Time, which is the total time of the FROSch preconditioner; see Figure 16 for visualization.

#Nodes	#Cores	#Subd.	#Subr.	$\kappa(M^{-1}K)$	Iter	Total Time	Solver Time	Setup Time	Krylov Time
32 MPI ranks per Theta node, 2 OpenMP threads per rank									
54	3 456	1 728	4	66.94	69	16.96 s	14.41 s	9.97 s	4.44 s
250	16 000	8 000	16	87.56	80	23.45 s	20.95 s	14.45 s	6.50 s
432	27 648	13 824	27	87.10	83	28.75 s	26.07 s	18.93 s	7.14 s
1 024	65 536	32 768	64	83.19	82	45.81 s	42.37 s	29.66 s	12.71 s
3 456	221 184	110 512	216	101.28	93	78.96 s	72.97 s	61.88 s	10.09 s

TABLE 12

More detailed timings on Theta. Two OpenMP threads per rank. One OpenMP thread per core. Ini. + Comp. = Setup. We have marked in **bold** *the (still remaining) scalability problem.*

#Nodes	#Cores	#Subd.	#Subr.	Size K_0	Size K_{00}	Init. Time	Comp. Time	Comm. K_0 Time	Fact. Time
32 MPI ranks per Theta node, 2 OpenMP threads per rank									
54	3 456	1 728	4	7 986	6	1.63 s	8.34 s	0.51 s	3.37 s
250	16 000	8 000	16	41 154	156	2.37 s	12.08 s	2.23 s	3.74 s
432	27 648	13 824	27	73 002	360	2.45 s	16.48 s	5.09 s	3.94 s
1 024	65 536	32 768	64	97 200	300	3.14 s	**26.52 s**	**13.36 s**	4.14 s
3 456	221 184	110 512	216	622 938	5 106	8.24 s	**53.65 s**	**35.41 s**	5.75 s

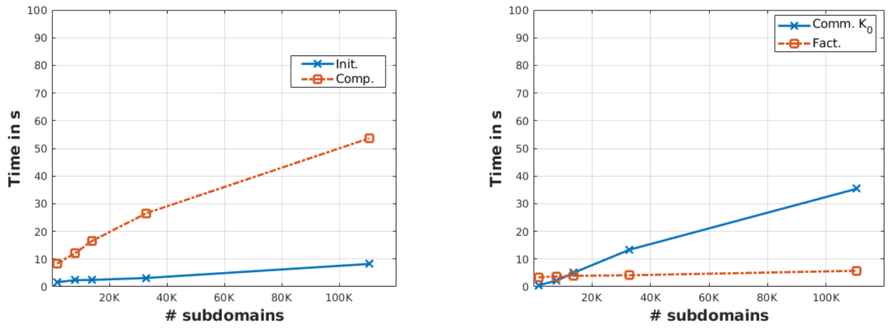

FIG. 15. *Time to initialize and to compute the FROSch preconditioner (left). Main time consuming parts of the compute time (Comp. Time): time to communicate the coarse matrix to the coarse solver (Comm. K_0 Time), and the numerical factorization (Fact. Time) (right). See Table 12 for the data.*

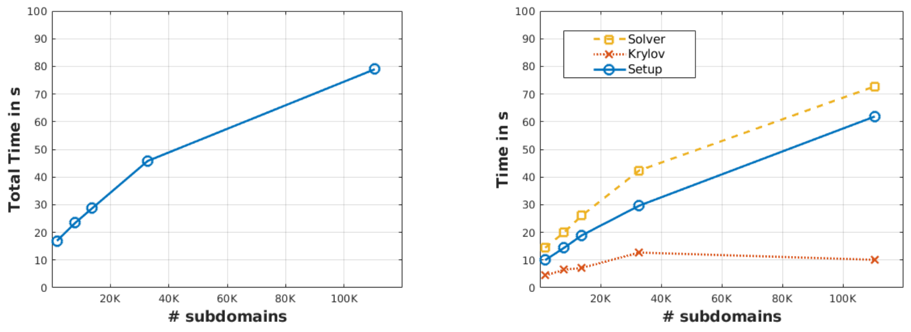

FIG. 16. *Weak parallel scalability on the Theta supercomputer with* 32 *MPI ranks per node and two OpenMP threads per rank. By Solver Time we denote the sum of Setup Time and Krylov Time, which is the total time of the FROSch preconditioner; see Table* 11 *for the data.*

timings indicate that we may have to revisit the implementation of the coarse communication described in subsection 5.4, for instance, calling a different MPI communication pattern within the Tpetra import and export methods. Since the same operation is at the order of one second on SuperMUC-NG, we plan to perform tests on other hardware as well.

7. Conclusion.

7. Conclusion. We have presented three-level overlapping Schwarz preconditioners of the GDSW and RGDSW types for three-dimensional linear elasticity. Their implementation in the FROSch software, which is part of Trilinos, uses the recursive application of the preconditioner to the coarse problem. We have then shown that in the recursive construction of the preconditioner the nullspace of the linear elastic operator, including the rotations, can be interpolated without using additional, explicit geometric information; moreover, as discussed in subsection 5.3, the same arguments are also valid for arbitrary nullspaces.

Using pure MPI parallelization on the Theta supercomputer, i.e., 64 ranks per node, revealed performance problems for FROSch for computations using about 100000 MPI ranks and more—presumably a result of MPI congestion, despite the hierarchical communication pattern. Using hybrid MPI/OpenMP, the onset of the communication problem can be delayed. However, the weak parallel scalability of the hybrid MPI/OpenMP parallel three-level FROSch solver using up to 220 000 cores of Theta is still not perfect. The construction of the coarse operator has to be revisited to improve the scalability on Theta further. Interestingly, this issue is not observed on SuperMUC-NG, where very good parallel scalability is obtained.

Acknowledgments. The authors also gratefully acknowledge the Gauss Centre for Supercomputing e.V. for computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre. The authors would like to acknowledge the 2021 ALCF Computational Performance Workshop of the Argonne Leadership Computing Facility, Argonne National Laboratory, USA. The authors also acknowledge the compute cluster (DFG project 397252409) of the Faculty of Mathematics and Computer Science of Technische Universität Bergakademie Freiberg, operated by the university computing center (URZ).

REFERENCES

[1] *Reference Documentation for deal.*ii *Version* 9.3.0 *- The Step-*8 *Tutorial Program*, 2021, https://www.dealii.org/current/doxygen/deal.II/step_8.html.

[2] *Trilinos Public Git Repository*, 2021, https://github.com/trilinos/trilinos.

[3] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells, *The deal.*II *finite element library: Design, features, and insights*, Comput. Math. Appl., 81 (2021), pp. 407–422, https://doi.org/10.1016/j.camwa.2020.02.022.

[4] S. Badia, A. F. Martín, and J. Principe, *Multilevel balancing domain decomposition at extreme scales*, SIAM J. Sci. Comput., 38 (2016), pp. C22–C52, https://doi.org/10.1137/15M1013511.

[5] A. H. Baker, A. Klawonn, T. Kolev, M. Lanser, O. Rheinbach, and U. M. Yang, *Scalability of classical algebraic multigrid for elasticity to half a million parallel tasks*, in Software for Exascale Computing - SPPEXA 2013-2015, H.-J. Bungartz, P. Neumann, and W. E. Nagel, eds., Springer, Cham, 2016, pp. 113–140, https://doi.org/10.1007/978-3-319-40528-5_6.

[6] A. H. Baker, T. V. Kolev, and U. M. Yang, *Improving algebraic multigrid interpolation operators for linear elasticity problems*, Numer. Linear Algebra Appl., 17 (2010), pp. 495–517, https://doi.org/https://doi.org/10.1002/nla.688.

[7] M. Bollhöfer, O. Schenk, R. Janalik, S. Hamm, and K. Gullapalli, *State-of-the-art sparse direct solvers*, in Parallel Algorithms in Computational Science and Engineering, A. Grama and A. H. Sameh, eds., Springer, Cham, 2020, pp. 3–33, https://doi.org/10.1007/978-3-030-43736-7_1.

[8] C. Burstedde, L. C. Wilcox, and O. Ghattas, *p*4*est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM J. Sci. Comput., 33 (2011), pp. 1103–1133, https://doi.org/10.1137/100791634.

[9] H. Carter Edwards, C. R. Trott, and D. Sunderland, *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*, J. Parallel Distrib. Comput., 74 (2014), pp. 3202–3216, https://doi.org/10.1016/j.jpdc.2014.07.003.

[10] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, *Run-to-run variability on Xeon Phi based Cray xc systems*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, Association for Computing Machinery, New York, NY, 2017, https://doi.org/10.1145/3126908.3126926.

[11] C. R. Dohrmann, A. Klawonn, and O. B. Widlund, *Domain decomposition for less regular subdomains: Overlapping Schwarz in two dimensions*, SIAM J. Numer. Anal., 46 (2008), pp. 2153–2168.

[12] C. R. Dohrmann, A. Klawonn, and O. B. Widlund, *A family of energy minimizing coarse spaces for overlapping Schwarz preconditioners*, in Domain Decomposition Methods in Science and Engineering XVII, Lecture Notes in Comput. Sci. Eng. 60, Springer, Berlin, 2008, pp. 247–254.

[13] C. R. Dohrmann and O. B. Widlund, *On the design of small coarse spaces for domain decomposition algorithms*, SIAM J. Sci. Comput., 39 (2017), pp. A1466–A1488, https://doi.org/10.1137/17M1114272.

[14] P. Fuentes, E. Vallejo, C. Camarero, R. Beivide, and M. Valero, *Network unfairness in dragonfly topologies*, J. Supercomput., 72 (2016), pp. 4468–4496, https://doi.org/10.1007/s11227-016-1758-z.

[15] GWT-TUD GmbH, *Vampir* 9 *User Manual*, Dresden, Germany, 2019, https://vampir.eu.

[16] W. Hackbusch, *Multigrid Methods and Applications*, Springer Ser. Comput. Math. 4, Springer-Verlag, Berlin, 1985, https://doi.org/10.1007/978-3-662-02427-0.

[17] A. Heinlein, C. Hochmuth, and A. Klawonn, *Fully Algebraic Two-Level Overlapping Schwarz Preconditioners for Elasticity Problems*, Tech. Rep., Universität zu Köln, 2019, https://kups.ub.uni-koeln.de/10441/.

[18] A. Heinlein, A. Klawonn, S. Rajamanickam, and O. Rheinbach, *FROSch: A fast and robust overlapping Schwarz domain decomposition preconditioner based on Xpetra in Trilinos*, in Domain Decomposition Methods in Science and Engineering XXV, R. Haynes, S. MacLachlan, X.-C. Cai, L. Halpern, H. H. Kim, A. Klawonn, and O. Widlund, eds., Springer, Cham, 2020, pp. 176–184, https://doi.org/10.1007/978-3-030-56750-7_19.

[19] A. Heinlein, A. Klawonn, and O. Rheinbach, *A parallel implementation of a two-level overlapping Schwarz method with energy-minimizing coarse space based on Trilinos*, SIAM J. Sci. Comput., 38 (2016), pp. C713–C747, https://doi.org/10.1137/16M1062843.

[20] A. Heinlein, A. Klawonn, O. Rheinbach, and F. Röver, *A three-level extension of the GDSW overlapping Schwarz preconditioner in two dimensions*, in Advanced Finite Element Methods with Applications: Selected Papers from the 30th Chemnitz Finite Element Symposium 2017, Springer, Cham, 2019, pp. 187–204, https://doi.org/10.1007/978-3-030-14244-5_10.

[21] A. Heinlein, A. Klawonn, O. Rheinbach, and F. Röver, *A three-level extension of the GDSW overlapping Schwarz preconditioner in three dimensions*, in Domain Decomposition Methods in Science and Engineering XXV, Springer, Cham, 2020, pp. 185–192, https://doi.org/10.1007/978-3-030-56750-7_20.

[22] A. Heinlein, A. Klawonn, O. Rheinbach, and F. Röver, *A Three-Level Extension for Fast and Robust Overlapping Schwarz (FROSch) Preconditioners with Reduced Dimensional Coarse Space*, Tech. report, TU Bergakademie Freiberg, February 2021, https://tu-freiberg.de/fakult1/forschung/preprints, submitted to the 26th International Conference on Domain Decomposition Methods.

[23] A. Heinlein, A. Klawonn, O. Rheinbach, and O. B. Widlund, *Improving the parallel performance of overlapping Schwarz methods by using a smaller energy minimizing coarse space*, in Domain Decomposition Methods in Science and Engineering XXIV, Springer, Cham, 2018, pp. 383–392.

[24] A. Heinlein, M. Perego, and S. Rajamanickam, *Frosch Preconditioners for Land Ice Simulations of Greenland and Antarctica*, Technical report, Universität zu Köln, January 2021, https://kups.ub.uni-koeln.de/30668/.

[25] A. Heinlein, O. Rheinbach, and F. Röver, *Choosing the subregions in three-level FROSch preconditioners*, in Proceedings of the 14th WCCM-ECCOMAS Congress, 2020, http://dx.doi.org/10.23967/wccm-eccomas.2020.084.

[26] P. Jolivet, F. Hecht, F. Nataf, and C. Prud'homme, *Scalable domain decomposition preconditioners for heterogeneous elliptic problems*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, New York, NY, 2013, ACM, pp. 80:1–80:11, https://doi.org/10.1145/2503210.2503212.

[27] B. Kiefer, O. Rheinbach, S. Roth, and F. Röver, *Variational methods and parallel solvers in chemo-mechanics*, PAMM. Proc. Appl. Math. Mech., 20 (2021), e202000272, https://doi.org/10.1002/pamm.202000272.

[28] A. Klawonn and O. Rheinbach, *Inexact FETI-DP methods*, Internat. J. Numer. Methods Engrg., 69 (2007), pp. 284–307, https://doi.org/10.1002/nme.1758.

[29] F. Kong and X.-C. Cai, *A highly scalable multilevel Schwarz method with boundary geometry preserving coarse spaces for 3D elasticity problems on domains with complex geometry*, SIAM J. Sci. Comput., 38 (2016), pp. C73–C95, https://doi.org/10.1137/15M1010567.

[30] J. Mandel, B. Sousedík, and C. R. Dohrmann, *Multispace and multilevel BDDC*, Computing, 83 (2008), pp. 55–85.

[31] L. F. Pavarino and S. Scacchi, *Parallel multilevel Schwarz and block preconditioners for the bidomain parabolic-parabolic and parabolic-elliptic formulations*, SIAM J. Sci. Comput., 33 (2011), pp. 1897–1919, https://doi.org/10.1137/100808721.

[32] S. Scacchi, *A hybrid multilevel Schwarz method for the bidomain model*, Comput. Methods Appl. Mech. Engrg., 197 (2008), pp. 4051–4061, https://doi.org/10.1016/j.cma.2008.04.008.

[33] J. Šístek, B. Sousedík, J. Mandel, and P. Burda, *Parallel implementation of multilevel BDDC*, in Proceedings of the 9th European Conference on Numerical Mathematics and Advanced Applications, Leicester, UK, 2011.

[34] A. Toselli and O. Widlund, *Domain Decomposition Methods—Algorithms and Theory*, Springer Ser. Comput. Math. 34, Springer-Verlag, Berlin, 2005.

[35] X. Tu, *Three-level BDDC in two dimensions*, Internat. J. Numer. Methods Engrg., 69 (2007), pp. 33–59, https://doi.org/10.1002/nme.1753.

[36] Xpetra Project Team, *The Xpetra Project Website*, https://trilinos.github.io/xpetra.html.

[37] Zoltan2 Project Team, *The Zoltan2 Project Website*, https://trilinos.github.io/zoltan2.html.