Proceedings of the Seventh International Conference on
Parallel, Distributed, GPU and Cloud Computing for Engineering
Edited by: P. Iványi, F. Magoulès and B.H.V. Topping
Civil-Comp Conferences, Volume 4, Paper 2.1
Civil-Comp Press, Edinburgh, United Kingdom, 2023
doi: 10.4203/ccc.4.2.1
©Civil-Comp Ltd, Edinburgh, UK, 2023

Automated Persistence of Subdomain Calculations in Finite Element Domain Decomposition

I. Kucuk[1] and M. Yilmaz[2]

[1] Graduate School, Istanbul Technical University Istanbul, Turkey
[2]Department of Civil Engineering, Istanbul Technical University Istanbul, Turkey

Abstract

This paper presents a domain decomposition framework with self-registering and reusable subdomains for finite element (FE) applications. The framework enhances traditional FE applications written in Python by automatically decomposing the system into subdomains and enabling the storage and reuse of existing subdomain solutions, significantly reducing the computational time for systems with minor changes. The proposed framework is evaluated through a solid mechanics application to assess its contribution to the analysis speed.

Keywords: domain decomposition, finite element method, Python, autosaving strategy, domain hash.

1 Introduction

Domain decomposition methods have gained popularity in finite element (FE) analysis due to their suitability for parallel computing and their ability to efficiently store and reuse subdomain solutions. By dividing the boundary value problem into smaller boundary value problems in subdomains, these methods facilitate parallel processing on different CPUs and provide the flexibility to implement existing subdomain solutions for systems with minor changes, thereby saving considerable computational time [1].

The concept of domain decomposition can be traced back to the seminal work of [2], who introduced the idea of overlapping subdomains to solve elliptic partial differential equations. H.A. Schwarz's alternating method laid the foundation for later developments in domain decomposition techniques, such as the additive and multiplicative Schwarz methods [3 and 4]. These methods have since been adapted to various applications, including fluid dynamics, structural mechanics, and electromagnetics [5 and 6].

With the advent of high-performance computing, domain decomposition methods have been widely used to leverage the power of parallel computing architectures. By dividing the global problem into smaller, independent subproblems, these techniques allow for concurrent execution on multiple processors, leading to significant reductions in computation time [7]. Parallel domain decomposition methods can be broadly categorized into two classes: overlapping methods, where subdomains share common nodes or elements, and non-overlapping methods, which involve the partitioning of the original domain into disjoint subdomains [5]. Recent advancements in domain decomposition methods have focused on enhancing the parallelization capabilities and improving the reuse of subdomain solutions [8 and 9].

Domain decomposition techniques have also been employed as preconditioners to enhance the convergence of iterative solvers for linear systems arising from FE discretization [10 and 11]. Preconditioning involves transforming the original linear system to accelerate the convergence of iterative methods, and domain decomposition-based preconditioners have been shown to be particularly effective for large-scale problems [6]. Various software frameworks have been developed to facilitate the implementation of domain decomposition methods for parallel computing [12 and 13]. These frameworks typically provide a high-level abstraction of the underlying parallelism and allow users to focus on the mathematical formulation of their problems.

While there is a not a direct study that specifically focuses on saving and reusing subdomain solutions (as far as the authors know), several studies and implementations indirectly address this concept or incorporate similar ideas. [14] presented a parallel FETI-DP (Finite Element Tearing and Interconnecting - Dual Primal) method for adaptively refined finite element meshes. While the primary focus is on the adaptive mesh refinement and the efficient parallelization of the FETI-DP method, the concept of reusing subdomain solutions is implicitly considered when the mesh is refined locally, and the previous subdomain solutions can be used as initial guesses for the iterative solver. [15] presented an adaptive multilevel method for high-order finite element methods conforming to H1 and H(curl) spaces. The method focuses on local mesh refinement and local smoothing techniques to improve the convergence of iterative solvers. Although these studies do not concentrate specifically on saving and reusing subdomain solutions, they involve concepts such as local mesh refinement and adaptive parallel methods that implicitly require the reuse of subdomain solutions. Incorporating the saving and reusing of subdomain solutions into these methods and frameworks could lead to further computational efficiency improvements [16].

The proposed domain decomposition framework interfaces with traditional FE applications written in Python by adding a few straightforward definitions to the end-user code. Automatic decomposition of the system into subdomains, registration, and reuse of subdomain solutions are performed within the framework.

To register and reuse subdomain solutions, a unique hash string representing the corresponding subdomain is created. This involves traversing all properties of individual items that constitute the subdomain, such as element and node properties and boundary conditions. Special Python decorators have been developed to allow the end-user to introduce variables of the functions in the end-user code to the decomposition framework. This streamlines the process of registering and reusing subdomain solutions. Standard components that automatically decompose end-user-defined FE objects into subdomains are also presented within the scope of the study.

The proposed framework is evaluated using a sample solid mechanics application to determine its contribution to the analysis speed. The evaluation demonstrates the potential benefits of the framework in terms of computational efficiency, particularly for systems with minor changes that do not require a complete rework of the problem. In this context, the proposed framework's ability to store and reuse subdomain solutions can be seen as an extension of the preconditioning concept, allowing for further improvements in computational efficiency.

An outline of the study is described as follows. Section 2 discusses general formulations of domain decomposition. In Section 3, autosaving subdomain solutions is investigated. Section 4 offers an illustrative FE implementation using the proposed framework and Section 5 evaluates the speed benchmarks for the given example. The paper concludes with a discussion of the presented approach in Section 6.

2 Domain Decomposition

A non-overlapping Domain Decomposition Method is used. The assumed linear Finite Element Analyses (FEA) model is given in Equation (1).

$$[K]\mathbf{U} = \mathbf{B} + \mathbf{S} + \mathbf{P} \tag{1}$$

In the model, $[K]$ is the Stiffness Matrix, $\mathbf{U}$ is the Degree of Freedom Vector, $\mathbf{B}$ and $\mathbf{S}$ are the external body force vector and external surface load vectors respectively, and $\mathbf{P}$ is the nodal force vector. Depending on the characteristics of the system to be calculated, there may be some additional external load vectors outside of and (e.g., temperature difference load vector or prestress load vector). However, within the scope of this study, it will be accepted that the right-hand side of the equation is only in the presented form.

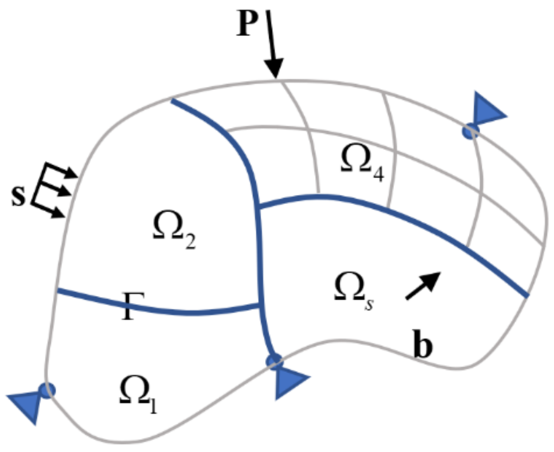

Figure 1: Subdomains and master interface (bold lines + nodes with Dirichlet boundary conditions). **P**: concentrated forces, **s**: surface forces, **b**: body forces.

The first step of the domain decomposition is to decompose the problem domain into subdomains Ω_s as depicted in Figure 1. Doing so reveals interface nodes (bold lines in the figure) which is called the master interface. In the second step, all subdomain equations are expressed in terms of two different variable groups as;

$$\begin{bmatrix} K_{11}^s & K_{12}^s \\ K_{21}^s & K_{22}^s \end{bmatrix} \begin{Bmatrix} \mathbf{u}_\Omega^s \\ \mathbf{u}_\Gamma^s \end{Bmatrix} = \begin{Bmatrix} \mathbf{q}_1^s \\ \mathbf{q}_2^s \end{Bmatrix} + \begin{Bmatrix} \mathbf{P}_\Omega^s \\ \mathbf{P}_\Gamma^s \end{Bmatrix} \tag{2}$$

where, $\mathbf{u}_\Omega^s$ represents the inner DOFs and $\mathbf{u}_\Gamma^s$ is the interface DOFs of the subdomain s. Note that, body and surface force vectors are also expressed as $\mathbf{q} = \mathbf{B} + \mathbf{S}$ for convenience. In the third step, Equation (2) is written in its open form as follows;

$$K_{11}^s \mathbf{u}_\Omega^s + K_{12}^s \mathbf{u}_\Gamma^s = \mathbf{q}_1^s + \mathbf{P}_\Omega^s \tag{3}$$

$$K_{21}^s \mathbf{u}_\Omega^s + K_{22}^s \mathbf{u}_\Gamma^s = \mathbf{q}_2^s + \mathbf{P}_\Gamma^s \tag{4}$$

Inner DOFs ($\mathbf{u}_\Omega^s$) can be eliminated with the help of Equation (5).

$$\mathbf{u}_\Omega^s = \left[K_{11}^s\right]^{-1} \left(\mathbf{q}_1^s + \mathbf{P}_\Omega^s - K_{12}^s \mathbf{u}_\Gamma^s\right) \tag{5}$$

The resulting subdomain composition is obtained in the form of Equation (6).

$$\left[K_{22}^s - K_{21}^s \left[K_{11}^s\right]^{-1} K_{12}^s\right] \mathbf{u}_\Gamma^s = \mathbf{q}_2^s - K_{21}^s \left[K_{11}^s\right]^{-1} \left(\mathbf{q}_1^s + \mathbf{P}_\Omega^s\right) + \mathbf{P}_\Gamma^s \tag{6}$$

Equation (6) can be written in compact form as;

$$K_\Gamma^s \mathbf{u}_\Gamma^s = \mathbf{q}_\Gamma^s + \mathbf{P}_\Gamma^s \tag{7}$$

where the following definitions hold.

$$K_\Gamma^s = \left[K_{22}^s - K_{21}^s \left[K_{11}^s \right]^{-1} K_{12}^s \right], \qquad \mathbf{q}_\Gamma^s = \mathbf{q}_2^s - K_{21}^s \left[K_{11}^s \right]^{-1} \left(\mathbf{q}_1^s + \mathbf{P}_\Omega^s \right) \tag{8}$$

3 Decomposition and Autosaving Subdomain Solutions

The process involves identifying subdomains within the main domain from an existing FE implementation. The classes included within the decomposition framework is summarized as follows.

- **Container**: Provided with the user-defined nodes and elements, this class creates the subdomains with an effective partitioning strategy while upgrading the given node and element with proper properties and methods to support for the decomposition with autosaving capabilities.

- **Decompose**: This class performs decomposition operations on the detected subdomains. These operations include;
    - Identification of the degree of freedoms for both the master interface and the corresponding subdomains.
    - Generating and solving the master interface equation of the system.

- **Subdomain**: This class performs subdomain equations. Basic functionality of the class is;
    - Generating hash string for the subdomains.
    - Constructing subdomain equations based on Equation (7).
    - Saving and retrieving back the relevant element and node matrices defined in Equation (8).

- **hashify**: A Python decorator that is designed to allow users to customize the hashing process for their user-defined functions.

The "Container" class is using a common strategy in computational geometry called spatial partitioning, specifically a technique called a quadtree [17, 18, 19 and 20], which is used to partition a space to make operations like search, insertion, deletion more efficient. The decision on how to split the domain is based on the geometry (length and width) of the domain. Container class is being used to group nodes or points in a certain spatial domain. Figure 2. demonstrates an example subdomain division performed by the Container class.

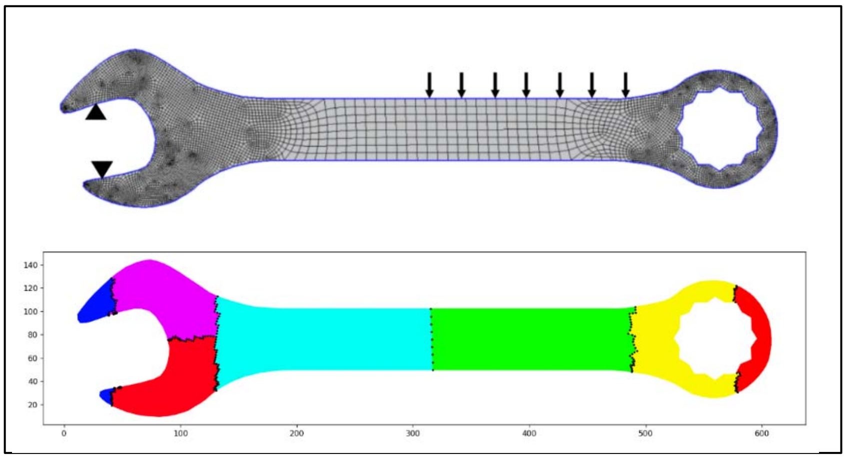

Figure 2: Example subdomain creation for a 2D plane stress mesh with 4 node quadrilateral elements. (Node Count: 6590, Element Count: 6252).

The "Decompose" class creates subdomain objects based on element groups provided by the Container. The pseudo-implementation of the Decompose class is demonstrated in Figure 3. The "solve" method is a function within the "Decompose" class that is designed to solve the master interface equations (see Equation (7)).

```
class Decompose:
  def __init__(self, nodes, elements, maxPointCount):
    - Creates the container object.
      #self.container=Container(nodes=nodes,elements=elements,maxPointCount)
    - Determines Node and Element groups that is marked by the container object.
    - Create subdomain objects based on element groups.
  def solve(self):
      # Solve the master interface equations by requesting subdomain methods (see Figure 4).
```

Figure 3: Pseudo-implementation of the *Decompose* Class.

The “Subdomain” class identifies the inner nodes and the master interface nodes from the given element group, the latter being the boundary nodes that connect with other subdomains. The class includes several methods, each of which calculates a different aspect of the subdomain's properties, while utilizing a caching system based on hash lists to improve performance. If there's no change in the hash of the subdomain, indicating no alterations, these methods retrieve previously computed solutions, saving computational time. The pseudo-implementation of the Subdomain class is demonstrated in Figure 4.

```
class Subdomain:
    def __init__(self, elements):
        - Determine inner-nodes and master-interface-nodes of the corresponding element group.
        # The methods outlined below utilize hash lists provided by the end-user to detect any
alterations in the subdomain. These methods also store their returns based on the generated hash.
If there's a change in the hash, they will recalculate the subdomain matrices. Otherwise, they'll
retrieve the solutions from previously computed ones.
    def get_K_Sparse: # Assembled Element Stiffness Matrix of the subdomain (see Eq. (1)).
    def get_K_Subdomain: # Condansated Subdomain Stiffness Matrix (see Eq. (7))
    def get_B_Subdomain: # Body forces vector of the subdomain elements (see Eq. (1)).
    def get_S_Subdomain: # Surface forces vector of the subdomain elements (see Eq. (1)).
    def get_P_Subdomain: # Concentrated forces vector of the subdomain elements (see Eq. (1)).
```

Figure 4: Pseudo-implementation of the *Subdomain* Class.

Lastly, the *hashify* class has been formulated as a Python decorator, aimed at empowering the end-user to discern the dependent variables associated with any field function (FE matrices and vectors) that they've integrated into their FE framework. An exemplification of the application of the decorator is provided in the subsequent chapter. The actual implementation of the class is demonstrated in Figure 5.

```
class hashify:
    def __init__(self, target, hash):
        self.target = target                  # Registering target function name
        self.hash_list_function = hash   # Resigtering function dependency (variables) list

    def get_values_to_hash(self, elm):
        # Returning hashable type (tuple) of the dependency list
        return tuple(self. hash_list_function(elm))

    def __call__(self, original):
        # Registering self to the decorated function (to be reached by the Subdomain)
        original._hashify_ = self
        return original
```

Figure 5: Implementation of the *hashify* Class as a Python Decorator.

4 Example Implementation

In this section, a demonstrative application of the proposed decomposition framework to plane stress problem will be presented. The effects of the Autosaving feature on the solution times of the problems will be examined. A typical implementation of a FE plane stress problem is given in Figure 6 and in Figure 7 with node and element implementations respectively.

```
nodes = dict()
class Node:
    def __init__(node, id, X, Y):
        node.id = id
        node.X, node.Y = X, Y   # Coordinates
        node.rest = [0, 0]          # Support Conditions [0: free, 1: fixed]
        node.force = [0, 0]         # Concentrated Loads [Px, Py]
        node.disp = [0, 0]          # Support Settlements
        node.code = [-1, -1]        # Degree of Freedom Numbers
        nodes[id] = node            # Saving node to a dictionary

    # "hash_by" returns a list of variables to be used in hashing the corresponding node object.
    def  hash_by(node):
        return [*node.rest, *node.force, *node.disp]
```

Figure 6: An example user-defined Node class definition of the plane-stress element along with the implementation of the *hash_by* method.

As shown with the example, the proposed decomposition framework can detect all dependencies of the methods that form the finite element equations for the nodes and elements via user-defined *hashify* functions that returns the list of dependencies of the corresponding methods. Therefore, the framework can discern whether there are any changes in the equations for subdomains containing these elements, by combining the hash functions of the elements related to the subregions into a single hash. As can be understood from the example application, the additional interface required for the proposed decomposition framework has been designed to impose a minimum cost on the end-user in terms of coding effort.

5 Results

Table 1. presents data on how the autosave feature impacts the solution time in different decomposition states. Specifically, investigates the effect of changing system parameters such as the Stiffness Matrix ([K]), B, S, and P across different subdomains.

The considered cases are as follows.

Case 0: A second solution after autosave <u>without any change in the system</u>.
Case 1: A second solution with Stiffness Matrix changed <u>in all subdomains</u>.
Case 2: A second solution with B, S, P changed <u>in all subdomains</u>.
Case 3: A second solution with Stiffness Matrix changed <u>in a single subdomain</u>.
Case 4: A second solution with only P changed in <u>a single subdomain</u>.

```
elements = dict()
class Element:
    def __init__(elm, id, conn, E, p, h):
      elm.id = id
      elm.conn = [nodes[id] for id in conn]  # Connectivity Vector
      elm.E = E                              # Elasticity Modulus
      elm.p = p                              # Poisson’s Ratio
      elm.h = h                              # Thickness
      elm.boundaryForce = [[0] * 4]*2        # Boundary Forces (4 surface, 2 components).
      elm.volumeForce = [0, 0]               # Volume forces [bx, by]
      elements[id] = elm                     # Saving element to a dictionary.

   …  # Classical definitions related to finite elements (for example, material matrix, coordinate
vector, Jacobian, etc. These definitions have not been included in the code as they are not directly
related to the subject of the study).

  @hashify(target= “K”, hash=lambda elm: [elm.E, elm.p, elm.h,
                                         *[(n.X, n.Y, *n.rest)  for n in elm.conn]])
   def K(elm):  # Element Stiffness Matrix (8x8). Decorated with hashify.
     def dK(r, s):
       h = elm.h              # Thickness
       C = elm.C()            # Constitutive (material) matrix (depens on E and p)
       BM = elm.BM(r, s)      # Strain-displacement matrix (depends on node coordinates).
       J = elm.detJM(r, s)    # Jacobian Matrix (depends on node corrdinates)
       return h * BM.T @ C @ BM * J
     return IntegrateOn2DDomainWithGaussN2(dK)


  @hashify(target= “B”, hash=lambda elm: [*elm.volumeForce,
                                         *[(n.X, n.Y, *n.rest) for n in elm.conn]])
   def B(elm): # Element body-force (volume-force) vector (8x1). Decorated with hashify.
     def dB(r, s):
       bx, by = elm.volumeForce
       h = elm.h              # Thickness
       J = elm.detJM(r, s)    # Jacobian Matrix (depends on node corrdinates)
       SFV = SF(r, s)         # Shape functions (Shape function Vector)
       SF8 = np.concatenate((SFV, SFV))
       return h * J * SF8 * [bx, bx, bx, bx, by, by, by, by]
     return IntegrateOn2DDomainWithGaussN2(dB)


  @hashify(target= “S”, hash=lambda elm: [*elm.boundaryForce,
                                         *[(n.X, n.Y, *n.rest) for n in elm.conn]])
   def S(elm): # Element boundary-surface external load vector (8x1). Decorated with hashify.
     def dS(r, s, k):
       qx, qy = elm.boundaryForce[0][k], elm.boundaryForce[1][k]
       SFV = SF(r, s)         # Shape functions (Shape function Vector)
       # … calculations continue.
       return J * np.concatenate((SFV * qx, SFV * qy))
     return IntegrateOn2DBoundariesWithGaussN2(dS)
```

Figure 7: An example user-defined Element class definition of the plane-stress element with *hashify* functions defined on the top of element matrix and element vector definitions.

Single Domain: DOF Count: **51102** Solution time without decomposition: **7.926** sec							
Subdomain Count	Master Interface DOF Count	First Solution Time (sec)	Case 0 Time (sec)	Case 1 Time (sec)	Case 2 Time (sec)	Case 3 Time (sec)	Case 4 Time (sec)
2	306	8.657	0.671	7.239	6.137	4.054	0.777
3	408	8.807	0.699	7.408	6.183	4.016	0.825
4	510	8.909	0.703	7.855	6.254	2.134	0.808
8	918	9.168	0.796	7.945	6.586	1.301	0.915
14	1530	9.837	0.891	8.007	6.841	1.356	1.012
27	2454	10.110	1.050	8.663	7.668	1.548	1.169

Table 1: The impact of the autosave feature on solution time for different decomposition states.

As the number of subdomains increases, so does the initial solution time and the time for each case. This suggests that more complex systems (with more subdomains) take longer to solve. The solution times in Case 0 (autosave feature with no changes) are significantly less than the initial solution times, indicating that the autosave feature has a considerable positive effect on reducing solution time when no changes are made to the system. While this case (Case 0) might not always seem particularly valuable, it can indeed be useful under certain circumstances. For instance, if there is a need to replicate the solution on a different machine, this benchmark time provides a practical reference.

The data implies that localized changes in a single subdomain, as in Cases 3 and 4, significantly reduce solution times and improve system efficiency. Conversely, system-wide changes, as in Cases 1 and 2, don't noticeably enhance solution times, suggesting that they are less efficient than localized changes.

6 Conclusions and Contributions

The presented domain decomposition framework with autosaving of subdomain solutions offers significant advantages for finite element analysis applications. By facilitating the automatic decomposition of the system into subdomains and enabling the storage and reuse of existing subdomain solutions, the framework substantially reduces computational time and enhances the overall efficiency of the analysis process. The autosave feature dramatically enhances computation speed (up to tenfold) when system changes are localized. This feature might be strategically leveraged in scenarios where system alterations can be restricted to a single subdomain, leading to significant efficiency gains. Given the generality of the methods employed, future investigations could explore expanding this framework to accommodate applications beyond the realm of FE. In essence, any computational process involving grouped calculations could potentially benefit from the presented approach.

Acknowledgement

This study was carried out within the scope of an ongoing master's thesis conducted by the first author at Istanbul Technical University Graduate School and under the supervision of the second author.

References

[1] G.P. Nikishkov, “Basics of the domain decomposition method for finite element analysis. Mesh Partitioning Techniques and Domain Decomposition Methods, Editor: Magoulés, F., Saxe-Coburg Publications, Kippen, Stirling, 119-142, 2007.

[2] H.A. Schwarz, "Über einen Grenzübergang durch Alternierendes Verfahren", Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich, 15, 272-286, 1870.

[3] P.L. Lions, "On the Schwarz Alternating Method III: A Variant for Nonoverlapping Subdomains", in Third International Symposium on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA, SIAM, 202-223, 1988.

[4] M. Dryja and O.B. Widlund, "An Additive Variant of the Schwarz Alternating Method for the Case of Many Subregions," Technical Report 339, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1987.

[5] A. Quarteroni and A. Valli, Domain Decomposition Methods for Partial Differential Equations. Oxford University Press, 1999.

[6] A. Toselli and O. Widlund, Domain Decomposition Methods—Algorithms and Theory, Springer Series in Computational Mathematics, vol. 34, Springer-Verlag, 2005.

[7] B.F. Smith, P.E. Bjørstad, and W.D. Gropp, Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press, 1996.

[8] J. Mandel, C.R. Dohrmann, and R. Tezaur, "An algebraic theory for primal and dual substructuring methods by constraints," Appl. Numer. Math., 113, 224-238, 2017.

[9] M.J. Gander and L. Halpern, "Schwarz methods: To share or not to share?" in C. Farhat, X. Cai, and F. Magoulès, editors, Domain Decomposition Methods in Science and Engineering XXIV, vol. 125, Lecture Notes in Computational Science and Engineering, Springer, 209-217, 2018.

[10] P.E. Bjørstad and O. B. Widlund, "Iterative Methods for the Solution of Elliptic Problems on Regions Partitioned into Substructures," SIAM Journal on Numerical Analysis, 23/6, 1093-1120, 1986.

[11] J.H. Bramble, J. E. Pasciak, and J. Xu, "Parallel Multilevel Preconditioners," Mathematics of Computation, 55/ 191, 1-22, 1990.

[12] W. Gropp, D. Kaushik, D. Keyes, and B. Smith, "Toward realistic performance bounds for implicit CFD codes," in Proceedings of the 11th International Conference on Domain Decomposition Methods, DDM.org, 1997, pp. 233-244.

[13] V. Dolean, P. Jolivet, and F. Nataf, An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel Implementation. SIAM, 2015.

[14] A. Klawonn and O. Rheinbach, "A parallel FETI-DP method for adaptive finite element meshes," International Journal for Numerical Methods in Engineering, 71/4, 413-426, 2007.

[15] J. Janssen and G. Kanschat, "Adaptive multilevel methods with local smoothing for H1- and H(curl)-conforming high order finite element methods," SIAM Journal on Scientific Computing, 33/4, 2095-2114, 2011.

[16] S. Balay, W.D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object-oriented numerical software libraries," in E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Modern Software Tools in Scientific Computing, Birkhäuser, 163-202, 1997.

[17] H. Samet, Foundations of Multidimensional and Metric Data Structures. San Francisco, CA: Morgan Kaufmann Publishers, 2006.

[18] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, Computational Geometry: Algorithms and Applications. Berlin, Germany: Springer-Verlag, 2008.

[19] H. Samet, "The Quadtree and Related Hierarchical Data Structures," in ACM Computing Surveys (CSUR), New York, NY: Association for Computing Machinery, 16/2, 187-260, 1984.

[20] T. Akenine-Möller, E. Haines, and N. Hoffman, Real-Time Rendering. Boca Raton, FL: CRC Press, 2008.