Algorithms for Parallel Generic *hp*-Adaptive Finite Element Software

MARC FEHLING, Colorado State University, USA
WOLFGANG BANGERTH, Department of Mathematics and Department of Geosciences Colorado State University, USA

The *hp*-adaptive finite element method—where one independently chooses the mesh size (h) and polynomial degree (p) to be used on each cell—has long been known to have better theoretical convergence properties than either h- or p-adaptive methods alone. However, it is not widely used, owing at least in part to the difficulty of the underlying algorithms and the lack of widely usable implementations. This is particularly true when used with continuous finite elements.

Herein, we discuss algorithms that are necessary for a comprehensive and generic implementation of *hp*-adaptive finite element methods on distributed-memory, parallel machines. In particular, we will present a multistage algorithm for the unique enumeration of degrees of freedom suitable for continuous finite element spaces, describe considerations for weighted load balancing, and discuss the transfer of variable size data between processes. We illustrate the performance of our algorithms with numerical examples and demonstrate that they scale reasonably up to at least 16,384 message passage interface processes.

We provide a reference implementation of our algorithms as part of the open source library `deal.II`.

CCS Concepts: • **Mathematics of computing → Computations in finite fields**; Mathematical software;

Additional Key Words and Phrases: Parallel algorithms, *hp*-adaptivity, finite element methods, high performance computing

ACM Reference format:
Marc Fehling and Wolfgang Bangerth. 2023. Algorithms for Parallel Generic *hp*-Adaptive Finite Element Software. *ACM Trans. Math. Softw.* 49, 3, Article 25 (September 2023), 26 pages.
https://doi.org/10.1145/3603372

1 INTRODUCTION

In the *hp*-adaptive variation of the **Finite Element Method (FEM)** for the solution of partial differential equations, one adaptively refines the mesh (h-adaptivity) and independently also chooses the polynomial degree of the approximation on every cell (p-adaptivity). This method is by now

This article is dedicated to the memory of William F. Mitchell.

This work used compute resources provided by the Extreme Science and Engineering Discovery Environment (XSEDE) [Towns et al. 2014], which is supported by National Science Foundation grant ACI-1548562. M. Fehling's work was supported by the National Science Foundation under award OAC-1835673 as part of the Cyberinfrastructure for Sustained Scientific Innovation (CSSI) program. W. Bangerth's work was partially supported by the National Science Foundation under award OAC-1835673, by award DMS-1821210, and by award EAR-1925595.

Authors' addresses: M. Fehling, Department of Mathematics, Colorado State University, 1874 Campus Delivery, Fort Collins, CO 80523-1874; email: marc.fehling@colostate.edu; W. Bangerth, Department of Mathematics and Department of Geosciences Colorado State University, USA; email: bangerth@colostate.edu.

0098-3500/2023/09-ART25 $15.00
https://doi.org/10.1145/3603372

40 years old [Babuška and Dorr 1981] and, at least from a theoretical perspective, well understood [Babuška and Guo 1996; Guo and Babuška 1986a, b]. In particular, it is known that *hp*-adaptivity provides better accuracy per **Degree of Freedom (DoF)** than either the *h*- or *p*-adaptive methods alone; more specifically, it exhibits a convergence rate where the approximation error in many cases decreases *exponentially* with the number of unknowns N—that is, the error satisfies $e = O(s^{-N})$ for some $s > 1$ that may depend on the solution rather than an algebraic rate $e = O(N^{-\gamma})$ for some $\gamma > 0$. In other words, *hp*-adaptivity is *asymptotically superior* to *h*- or *p*-adaptivity alone.

Yet, *hp-adaptive methods are not widely used.* The reasons for this lack of use are probably debatable but surely include (i) that the literature provides many criteria by which to choose whether *h*- or *p*-refinement should be selected if the error on a cell is large, but that there is no consensus on which one is best, and (ii) a lack of widely usable implementations. For the first of these points, we refer to the comprehensive comparison in the work of Mitchell and McClain [2014]. Instead, in this contribution, we address the second point: the lack of widely available implementations.

A survey of the finite element landscape shows that there are few options for those who are interested in experimenting with *hp*-methods. Most of the open source distributed-memory parallel implementations of *hp*-adaptive methods available that we are aware of—specifically the ones in the libraries `PHAML` [Mitchell 2002], `PHG` [Zhang 2019], and `MoFEM` [Kaczmarczyk et al. 2020]—have not found wide use in the community and are not backed by large user and developer communities. To the best of our knowledge, other popular libraries like `FEniCS`/`FEniCSx` [Alnæs et al. 2015], `GetFEM` [Renard and Poulios 2020], and `FreeFEM++` [Hecht 2012] do not offer *hp*-adaptive methods at all or have only experimental support as is the case with `libMesh` [Kirk et al. 2006].

In other cases, such as the ones discussed in several works [Bey et al. 1996; Chalmers et al. 2019; Paszyński and Demkowicz 2006; Paszyński and Pardo 2011], the implementation of *hp*-methods is restricted to **Discontinuous Galerkin (DG)** methods; the same limitation also applies to the libraries `MFEM` [Anderson et al. 2021; Pazner and Kolev 2022] and `DUNE` [Bastian et al. 2021; Gersbacher 2016]. This case is relatively easy to implement because the construction of finite element spaces is purely local, on every cell independent of its neighbors. At the same time, discontinuous Galerkin (DG) methods are expensive—especially in three dimensions—because degrees of freedom (DoFs) are duplicated between neighboring cells, and the resulting large linear systems and corresponding memory consumption have hampered adoption of DG schemes in most applications outside the simulation of hyperbolic systems. As a consequence, although the use of DG methods for *hp*-adaptivity is a legitimate approach, there are many important use cases where continuous finite element spaces remain the method of choice.

Finally, let us mention publications [Jomo et al. 2017; Paszyński and Pardo 2011] that also demonstrate the use of *hp*-adaptive methods; these use distributed-memory parallelization, but use data structures for the mesh replicated on all processes, thus limiting scalability. An extension to distributed data structures, using a hierarchical construction of finite element spaces, is discussed in the work of Jomo [2021]. The `Hermes` library [Šolín et al. 2008] falls into a separate category, using shared-memory parallelism. Laszloffy et al. [2000] do present distributed-memory algorithms but only show scaling to 16 processors, whereas we are interested in much larger levels of parallelism. We are not aware of any commercial tools capable of using *hp*-methods, either for sequential or parallel computations.

As a consequence of our search for available implementations, and to the best of our knowledge, only the `deal.II` library [Arndt et al. 2021, 2022] appears to have generic support for *hp*-adaptive methods for a wide variety of finite elements, discontinuous or continuous, as discussed previously in detail in the work of Bangerth and Kayser-Herold [2009]. Still, `deal.II` has only recently begun to support *hp*-adaptive methods for parallel computations [Fehling 2020]. It is this specific gap that we wish to address in this contribution, by considering what algorithms are necessary

to implement *hp*-methods *on large parallel machines* using a distributed-memory model based on the **Message Passing Interface (MPI)**. The target for our work is the solution of 2D and 3D scalar- or vector-valued partial differential equations, using an arbitrary combination of finite elements, and scaling up to tens of thousands of processes and billions of unknowns.

More specifically, we identify and address the following three major challenges in this work:

- The development of a scalable algorithm to *uniquely enumerate DoFs* on meshes on which finite element spaces of different polynomial degrees may be associated with each cell. Simply enumerating all DoFs on a mesh turns out to be nontrivial already in distributed-memory implementations of *h*-adapted, unstructured meshes (as discussed in the work of Bangerth et al. [2012]), as well as for sequential implementations of the *hp*-method (see the work of Bangerth et al. [2007]), and it is no surprise that the combination of the two leads to additional complications.
- An efficient distribution of workload among all processes with *weighted load balancing*, since the workload per cell depends on its local number of DoFs and thus varies from cell to cell with *hp*-adaptive methods. We will present strategies on how to determine weights on each cell for this purpose.
- The ability to *transfer data of variable size* between *hp*-adapted meshes during repartitioning. In the *hp*-context, the amount of data stored per cell is proportional to the number of local DoFs and, again, varies between cells.
- An assessment of the parallel efficiency of the algorithms mentioned previously.

In this article, we first address the task of enumerating all DoFs in a distributed-memory setting in Section 2. We then present strategies for weighted load balancing in Section 3 and continue with ways to transfer data of variable size in Section 4. In Section 5, we illustrate the performance and scalability of our methods using numerical results obtained on the Expanse supercomputer [Strande et al. 2021], using up to 16,384 cores. We present conclusions in Section 6.

Code Availability. The algorithms we discuss in the remainder of this article are implemented and available in the open source library `deal.II`, version 9.4 [Arndt et al. 2021, 2022]. All functionality is available under the LGPL 2.1 license. That said, our discussions are not specific to `deal.II` and are generally applicable to any other finite element software. In particular, even though we will only show examples of quadrilateral or hexahedral meshes, our algorithms are readily applicable also to simplex or mixed meshes.

The two programs that implement the test cases of Section 5.1 and for which we show results in Sections 5.2 and 5.3 are available as part of the tool `hpbox` [Fehling 2022].

2 ENUMERATION OF DOFS

In the abstract, the FEM defines a finite-dimensional space V_h within which one seeks the discrete solution of a (partial) differential equation. In practice, one needs to construct a basis $\{\varphi_i\}_{i=0}^{N-1}$ for this space so that numerical solutions $u_h \in V_h$ can be expressed as expansions of the form $u_h(\mathbf{x}) = \sum_i U_i \varphi_i(\mathbf{x})$, where the U_i are the nodal coefficients of the expansion.

The basis functions of V_h are mathematically defined via *nodal functionals* [Brenner and Scott 2008], but for the purposes of this section, it is only important to know that each basis function is associated with either a vertex, an edge, a face, or the interior of a cell of a mesh. To enumerate the DoFs on an unstructured mesh, one therefore simply walks over all cells, faces, edges, and vertices, and, in a first step, allocates as much memory as is necessary to store the indices of DoFs associated with each of these entities, setting the index to an invalid value. In a second step, one then repeats the loop and assigns consecutive indices to each degree encountered that has not yet received a valid number.

Our goal herein is to come up with an algorithm that replicates this action for parallel computations if the data structures that represent the mesh and the indices of DoFs defined thereon are distributed across individual nodes of a parallel computer. Implicit in this goal is that we continue to want a single, global mesh along with a *global enumeration* of all DoFs—even though each process in this distributed-memory universe only sees its own small part of this distributed data structure. There are of course other ways to solve partial differential equations in parallel, most notably using domain decomposition methods in which each process has only a local enumeration of the DoFs located on that part of the mesh it owns (and potentially on one or more layers of "ghost cells"), plus a way to map its local indices to the local indices on neighboring processes at partition interfaces and on ghost cells. Yet, domain decomposition methods have somewhat fallen out of favor because they do not scale well to very large process counts, and have largely been replaced by methods that instead consider a global finite element space (with basis functions indexed globally) from which one then builds a single, global linear system that is stored in a distributed data structure spread across processes. It is this latter model we seek to support in our work, requiring a global enumeration of all DoFs with globally unique identifiers.

In the remainder of this section, our goal is to describe an algorithm that achieves this enumeration of DoFs in parallel for the *hp*-adaptive case. For context, let us first briefly outline how this is done for distributed, unstructured meshes when only one type of finite element is used (Section 2.1), followed by a description of the algorithm used for *hp*-adaptive methods on a single process (Section 2.2). In Section 2.3, we then present our new algorithm for parallel *hp*-adaptive methods, which can be seen as a combination and enhancement of the former two.

We do not cover details on handling hanging nodes and constraints in this article. It turns out that for the new algorithm, their handling does not require any change from the methods described in other works [Bangerth et al. 2012; Bangerth and Kayser-Herold 2009].

2.1 Enumerating DoFs on Distributed, Unstructured Meshes

In a parallel program where the mesh data structure is stored in distributed memory, the situation is complicated by the fact that each process only knows a subset of cells—namely, those cells that are "locally owned" along with a layer of "ghost cells." At the same time, we need to assign *globally unique* indices to all entities of the distributed mesh: at the end of the algorithm, each process must know the global indices of those DoFs that are located on this process's locally owned and ghost cells.

For the relatively simple case where the finite element is the same on each cell (no *p*-adaptivity), the index assignment is typically achieved by identifying a tie-breaking process that defines which process "owns" a mesh entity on the interface between the sub-domains of cells owned by individual processes (i.e., which of the adjacent processes owns a vertex, an edge, or a face on this interface). This process is then also the owner of the DoFs located on these entities. A possible tie-breaker is that the process with the smallest Message Passing Interface (MPI) rank is chosen as the owner of an entity on a subdomain interface.

Enumeration of DoFs then proceeds by each process enumerating the DoFs it owns, starting at zero. All of these indices are then shifted so that we obtain globally unique indices across processes. Next, each process sends the indices associated with locally owned cells to those processes that have these cells as ghost cells. Because processes may not yet know all DoF indices on the boundaries of locally owned cells at the time of this communication step, the exchange has to be repeated a second time to ensure that each process knows the full set of indices on both the locally owned cells as well as ghost cells, and all of the vertices, edges, and faces bounding these cells.

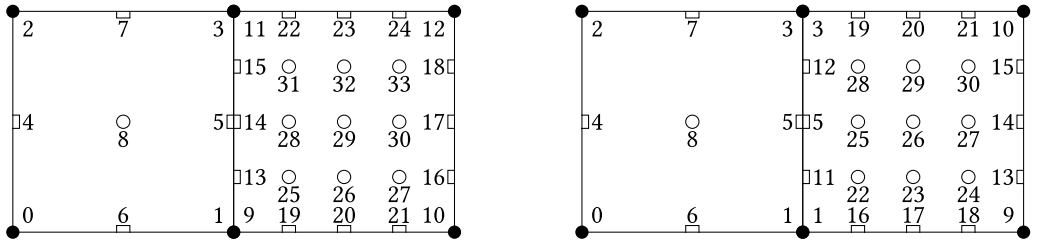


Fig. 1. Enumeration of DoF indices on a mesh with two cells on which the left cell uses a Q_2 (bi-quadratic) Lagrange element and the right cell uses a Q_4 (bi-quartic) element. We distinguish between support points on vertices (•), lines (□), and quadrilaterals (○). Left: Naive enumeration of DoFs. Continuity is ensured through constraints. Right: A better way in which we "unify" some DoFs.

A formal description of this algorithm—which consists of five stages—has been given in the work of Bangerth et al. [2012] and forms the basis of the discussions for the parallel *hp*-case in Section 2.3.

2.2 Enumerating DoFs in the Sequential *hp*-Context

In the *hp*-context, each cell $K \in \mathbb{T}$ of a triangulation or mesh $\mathbb{T}$ may use a different finite element. To make the notation that we use in the following concrete, let us assume that we want the global function space V_h be constructed so that the solution functions $u_h \in V_h$ satisfy $u_h|_K \in V_h(K)$, where $V_h(K)$ is the finite element space associated with cell K. Furthermore, let us assume that $V_h(K)$ can only be one from within a collection of spaces $\{\hat{V}_h^{(i)}\}_{i=0}^{I}$ defined on the reference cell $\hat{K}$ that are then mapped to cell K in the usual way—that is, $V_h(K) = \mathcal{M}_K \hat{V}_h$, where $\mathcal{M}_K$ is the operator that maps the finite element space from the reference cell to K; the details of this mapping are not of importance to us here. We denote the "active finite element index" on cell K by $a(K)$ (i.e., $V_h(K) = \mathcal{M}_K \hat{V}_h^{(a(K))}$). Each of the spaces $\hat{V}_h^{(i)}$ has a number of DoFs associated with each vertex, edge, face, and cell interior.

A trivial implementation of enumerating all DoFs would simply loop over all cells $K \in \mathbb{T}$ and enumerate all DoFs on both the cell K and its vertices, edges, and faces independently of the enumeration on neighboring cells. To do so requires storing multiple sets of indices of DoFs on vertices, edges, and faces, each set corresponding to one of the adjacent cells. This strategy would result in a global finite element space that is discontinuous between neighboring cells, but continuity can be restored by adding constraints that relate DoFs on neighboring cells.

The left panel of Figure 1 illustrates this approach. Here, each cell's DoFs are independently enumerated. Continuity of the solution is then restored by introducing identity constraints of the form $U_9 = U_1$, $U_{11} = U_3$, $U_{14} = U_5$, in addition to the more traditional "hanging node constraints" $U_{13} = \frac{3}{8}U_1 - \frac{1}{8}U_3 + \frac{3}{4}U_5$, $U_{15} = -\frac{1}{8}U_1 + \frac{3}{8}U_3 + \frac{3}{4}U_5$.

Although conceptually simple, this approach is wasteful, as it introduces many more DoFs than necessary, along with a large number of constraints. In the extreme case of using Q_1 (tri-linear) Lagrange elements on all cells of a uniformly refined 3D mesh, one ends up with approximately eight times as many DoFs, 7/8 of which are constrained. In actual test cases using *hp*-adaptivity, in Section 4.2 of the work of Bangerth and Kayser-Herold [2009], the authors report that these "unnecessary" DoFs can be up to 15% of the total number of DoFs in 3D.

To avoid this wastefulness, the algorithms described in the work of Bangerth and Kayser-Herold [2009] "unify" DoFs where possible during the enumeration phase. For example, in the case shown in Figure 1, the DoFs on shared vertices can be unified for the particular choice of elements adjacent to these vertices, as can be the DoF located on the common edge's midpoint. This leads to the

enumeration shown on the right side of the figure, for which we then only need to add constraints $U_{11} = \frac{3}{8}U_1 - \frac{1}{8}U_3 + \frac{3}{4}U_5$, $U_{12} = -\frac{1}{8}U_1 + \frac{3}{8}U_3 + \frac{3}{4}U_5$.

At the same time, it is clear that this "unification" step requires knowing about the global indices of DoFs on neighboring cells *during enumeration*, and this presents issues that need to be addressed in the parallel context if one of the cells adjacent to a vertex, edge, or face is a ghost cell. Furthermore, each process must know the active finite element index not only for its locally owned cells but also for ghost cells, before the enumeration can begin. We have to take into account all of these considerations in the extension of the algorithms of Bangerth and Kayser-Herold [2009] to the parallel context in the next section.

2.3 The Parallel *hp*-Case

Having discussed the fundamental algorithms necessary to globally enumerate DoFs in the context of both parallel unstructured meshes, and for the sequential *hp*-case, let us now turn to an algorithm that combines both of these features. As we will see, this algorithm turns out to be nontrivial.

2.3.1 Goals for the Parallel Algorithm. In developing such an enumeration algorithm, we are guided by the desire to come up with an enumeration that leads to a total number of DoFs that is independent of the number of processes. In other words, we do not want to treat vertices, edges, or faces that happen to lie on subdomain boundaries any different than if they were within the interior of a subdomain. We consider this an important feature to achieve scalable and predictable algorithms, and because it makes debugging problems easier. Furthermore, we would like to develop an algorithm that includes the "unification step" mentioned earlier to avoid generating too many trivial constraints.

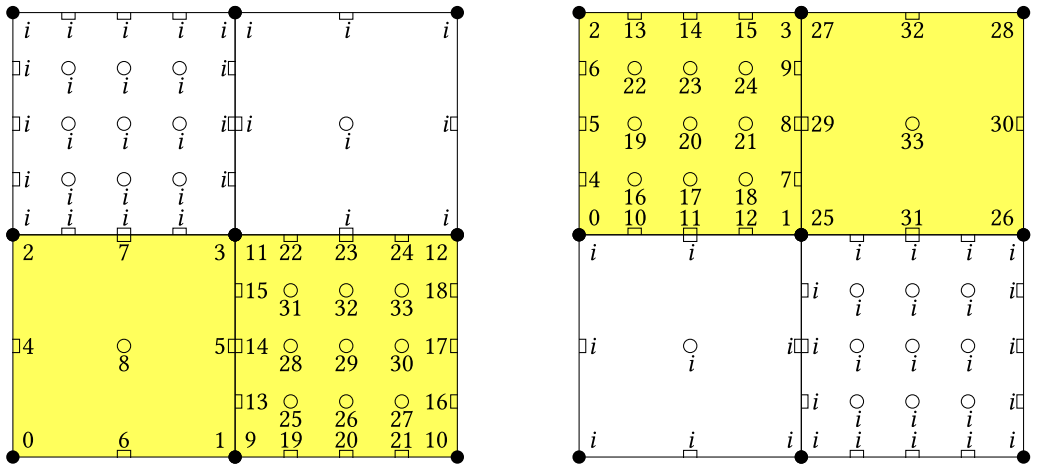
At the end of the algorithm, each parallel process needs to know the globally unique indices of all DoFs located on the locally owned cells as well as on ghost cells, including the outer vertices, edges, and faces of ghost cells beyond which the current process has no knowledge of whether and how the mesh continues.

Finally, we want this algorithm to have linear complexity in the number of cells or the number of DoFs. We achieve this by stating it as a fixed-length series of loops over all cells owned by each process and, if necessary, over all ghost cells on this process. Because each process only loops over its own cells, and because the number of DoFs per process is balanced (also see Section 3), we obtain an algorithm that we expect to scale optimally both strongly and weakly.

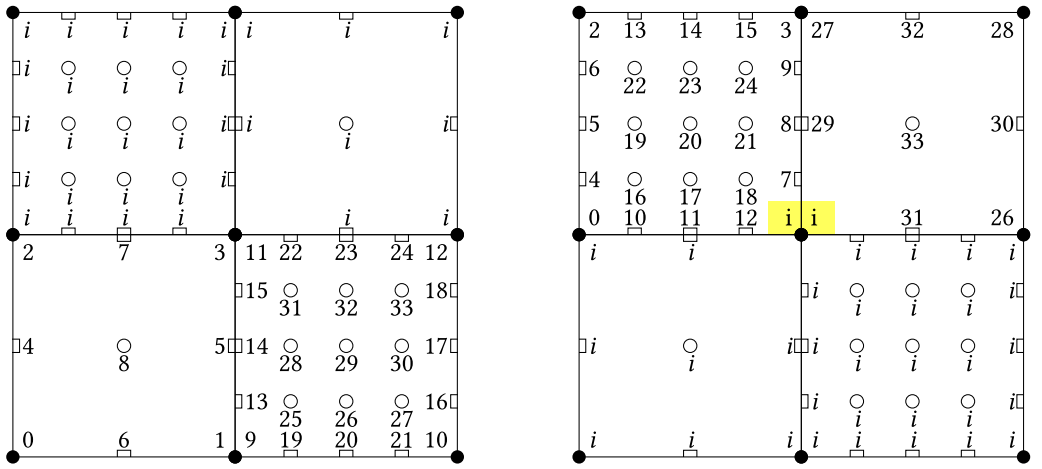
The algorithm that achieves all of this—see the following discussion—can be broken down into seven distinct stages. In addition to their description, we illustrate each stage in an example for which we consider a 2D mesh of four neighboring cells meeting at a central vertex. On this mesh, we use bi-quadratic (Q_2) Lagrange elements with 9 unknowns on the bottom left and top right cell, and bi-quartic (Q_4) elements with 25 unknowns on the remaining two cells. Furthermore, we assume that the partitioning algorithm has divided the mesh into two subdomains: subdomain zero contains the bottom two cells, and subdomain one the top two. This setup is shown in Figure 2, where we illustrate the progress of the enumeration algorithm. Each figure shows the view from process zero on the left, and from process one on the right.

We note that although we illustrate the workings of our algorithm using nodal Lagrange elements, nothing in this article is specific to these elements. Indeed, what matters is only how many DoFs an element has per vertex, edge, face, and cell interior, and the ability to identify the order of DoFs on an entity. As a consequence, where the figures show nodal DoFs as points on edges, one could also think of these as representing successively higher edge moments of modal elements.

2.3.2 Algorithm Inputs. The algorithm we describe in the following needs the following pieces of information as inputs:



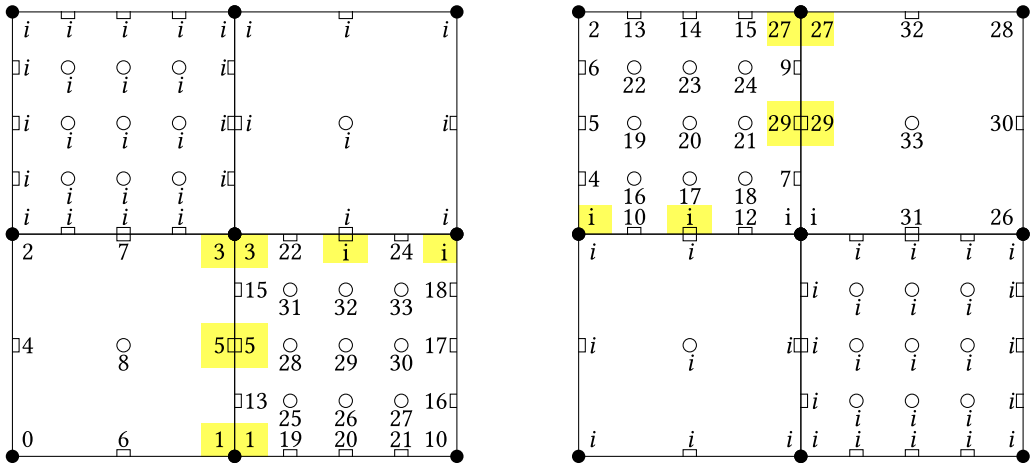
(After Stage 1) Local enumeration.



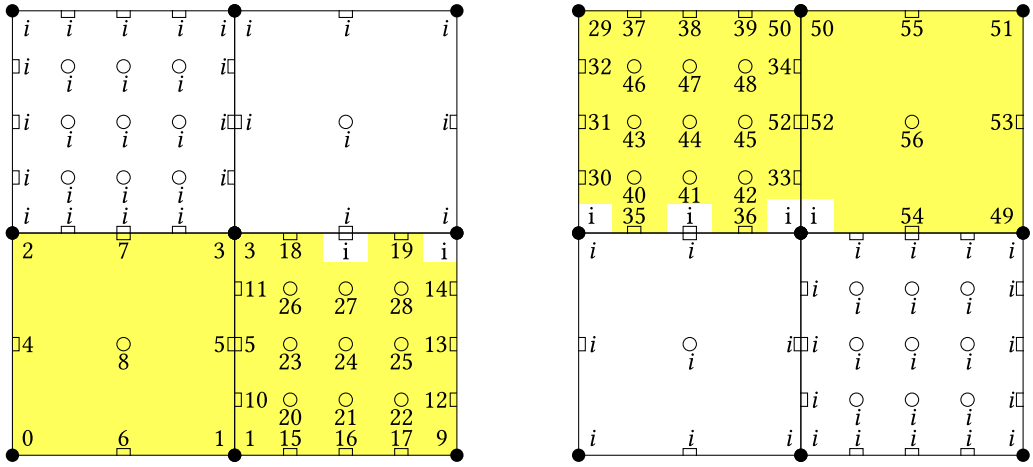
(After Stage 2) Tie-break.

Fig. 2. Exemplary application of our enumeration algorithm for DoFs. Changes made at each step are highlighted. The left diagram of each subfigure depicts the situation for process 0, whereas the right side shows the domain from the perspective of process 1. The top half of each subfigure constitutes subdomain 1, whereas the bottom cells are assigned to subdomain 0. DoFs on ghost cells are marked by italic indices.

- A set of cells K that constitute the *locally owned* and *ghost* cells, and information how neighboring cells are connected. The algorithm does not need to know where these cells are geometrically located in an ambient space—although this is of course important for the downstream application of the FEM—but only the topological connection of vertices, edges, and faces to the cells of which they are part of. The meshes we allow can be locally refined, with hanging nodes along edges. For practical reasons, we only allow a single hanging node per edge (i.e., we enforce a 2:1 ratio of neighboring cell sizes), although this is not a critical restriction for the algorithm we will show.
- A process must know to which process each of its ghost cells belongs. Since we identify subdomain ids with process ranks in a MPI universe, that means that we need to store the owning process's subdomain, or short owner, of all ghost cells.



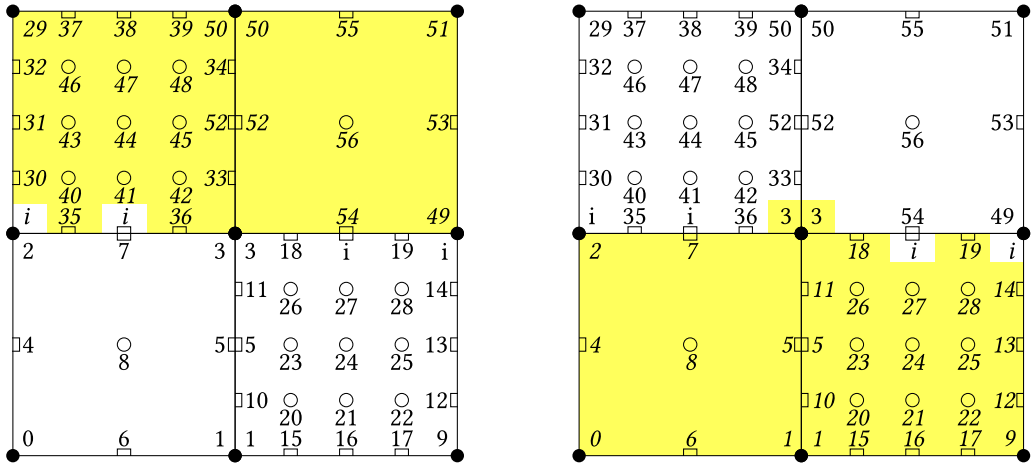
(After Stage 3) Unification.



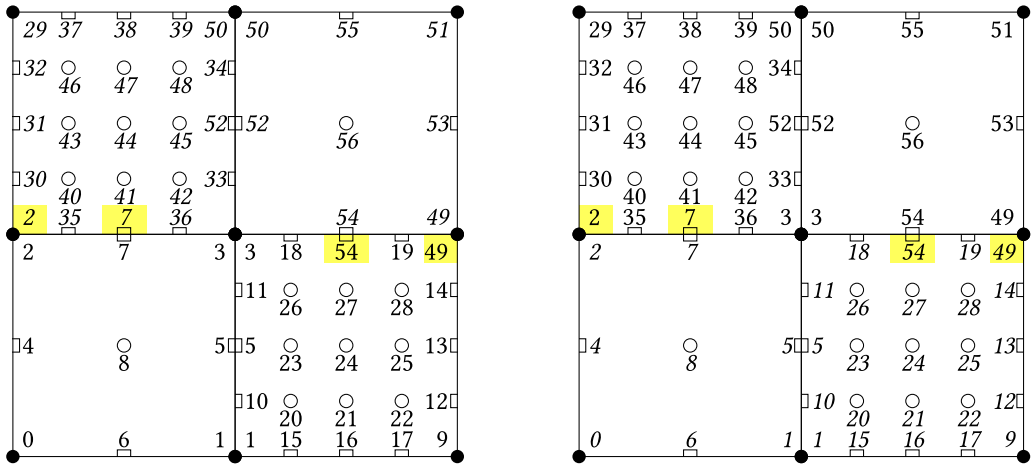
(After Stage 4) Global re-enumeration.

Fig. 2. Continued.

- Each cell on every process has an associated *global* identifier. This identifier is the same on all processes that store this cell, whether as part of their locally owned cells or as a ghost cell.
- For each locally owned or ghost cell, every process must know the active finite element index—that is, which element $\hat{V}_h^{a(K)}$ is in use on each cell K. Because the active finite element index is typically computed only on each process's locally owned cells, this information needs to be exchanged between processes before the start of the algorithm; as with any ghost exchange of information, this can efficiently be done through point-to-point communication.
- For each element in the collection $\{\hat{V}_h^{(i)}\}_{i=1}^{I}$, the algorithm needs to know how many DoFs this element has per vertex, edge, face, or cell interior. For example, for the Q_4 Lagrange element in 2D that we use in our illustrative example, there is one DoF per vertex, three per edge, and nine per cell interior.
- A data structure that can store the indices of DoFs located on each vertex, edge, face, and cell that is either locally owned or part of a ghost cell. Specifically, this implies that if one



(After Stage 5) Ghost exchange.



(After Stage 6) Merge on interfaces.

Fig. 2. Continued.

enumerates DoFs on one cell and sets their indices on that cell, that the indices of DoFs on vertices, edges, and faces are immediately also visible from neighboring cells without having to explicitly duplicate this information to other cells' index arrays. As discussed in the work of Bangerth and Kayser-Herold [2009], lower-dimensional entities (vertices, edges, and faces) have to be able to store multiple sets of indices, one for each of the finite element spaces used on the cells adjacent to the entity.

- For each pair of elements, the corresponding finite element implementations need to be able to identify whether two DoFs located on the same entity (vertex, edge, or face) can be unified. For example, our algorithm needs to be able to ask the combination of the Q_2 and Q_4 elements in the example shown in Figure 1 whether the single DoF each wants to store on a shared vertex can receive a single index, and whether the one Q_2 DoF on the shared edge can be unified with one of the three DoFs the Q_4 element wants to allocate on the common

edge. In the example of Figure 1, the answer is "yes" (DoFs 9, 11, and 14 in the left subfigure can be identified with DoFs 1, 3, 5, and this is done in the right subfigure). But this is not always the case, and consequently it is necessary to be able to query the adjacent finite elements about the possibility of DoF unification (e.g., DoFs 13 and 15 in the left subfigure cannot be unified with any DoF of the left cell, and similar cases also often happen when elements are defined with modal instead of nodal shape functions). The details of how an answer to such a query can be implemented are not relevant to our description here but are discussed at length in the work of Bangerth and Kayser-Herold [2009].

2.3.3 Description of the Algorithm. The algorithm we show consists of seven stages (plus an initial memory allocation and initialization stage), as detailed in the following. To make understanding it easier, we illustrate each step in Figure 2 for our model test case; note that the figure is continued over several pages. In our description, we follow the same nomenclature as Bangerth et al. [2012]. Specifically, we generally use an index p or q for subdomains (identified with process ranks in an MPI universe), and we denote the set of all *locally owned cells* on process p by $\mathbb{T}^p_{\text{loc}}$, the set of all *ghost cells* by $\mathbb{T}^p_{\text{ghost}}$, and the set of all *locally relevant cells* by $\mathbb{T}^p_{\text{rel}} = \mathbb{T}^p_{\text{loc}} \cup \mathbb{T}^p_{\text{ghost}}$. In the following description, the process index p is generically used to identify the "current" process—that is, the rank of the process that is executing the algorithm.

Our algorithm then proceeds in the following steps:

(0) *Initialization (without illustration):* Loop over all locally relevant cells $K \in \mathbb{T}^p_{\text{rel}}$, and on each of its vertices, edges, faces, and K itself, allocate enough space to store as many DoF indices as are necessary for the element identified by the active finite element index $a(K)$. If a neighboring element has already allocated space for the same active finite element index, then no additional space is necessary. In other words, for each entity within $\mathbb{T}^p_{\text{rel}}$, we need to allocate space for a map from the active finite element indices of adjacent cells to an array of indices of DoFs indices.

Once space is allocated, all DoF indices are set to an invalid value that we denote by i in the following (e.g., $i \coloneqq -1$).

Communication: This stage does not require any communication.

(1) *Partition-local enumeration:* Iterate over all locally owned cells $K \in \mathbb{T}^p_{\text{loc}}$. For each of the vertices, edges, faces, and the cell interior, assign valid DoF indices in ascending order, starting from zero, if indices have not already been assigned for an entity and the current $a(K)$.

Communication: This stage does not require any communication.

(2) *Tie-break:* Iterate over all locally owned cells $K \in \mathbb{T}^p_{\text{loc}}$. If a vertex, edge, or face that is part of K is also part of an adjacent ghost cell K'_{ghost} so that $a(K) = a(K'_{\text{ghost}})$, and if K'_{ghost} belongs to a subdomain of lower rank $q < p$, then invalidate all DoFs on this mesh entity by setting their index to the invalid value i.

Communication: This stage does not require any communication.

(3) *Unification*: Iterate over all locally owned cells $K \in \mathbb{T}^p_{\text{loc}}$. For all shared DoFs on vertices, edges, and faces to neighboring cells K' (locally owned or ghost), ask the elements corresponding to active finite element indices $a(K)$ and $a(K')$ whether some of the DoFs can be unified between the two elements. If K' is also a locally owned cell, perform the unification by replacing one index (or a set of indices) by the corresponding index of the other DoF to which it is unified. If K' is a ghost cell, and if the DoF on K needs to be unified with the corresponding one on K' (rather than the other way around), then set the index of the DoF on K to the invalid value i.

Communication: This stage does not require any communication.

At this point in the algorithm, each process knows which DoFs are owned by this process—namely, the ones on locally owned cells that are enumerated as anything other than i—although the final indices of these DoFs are not yet known.

(4) *Global re-enumeration*: Iterate over all locally owned cells $K \in \mathbb{T}^p_{\text{loc}}$ and re-enumerate those DoF indices in ascending order that have a valid value assigned, ignoring all invalid indices. Store the total number of all valid DoF indices on this subdomain as n_p. In a next step, shift all indices by the number of DoFs that are owned by all processes of lower rank $q < p$, or in other words, by $\sum_{q=0}^{p-1} n_q$. Computing this shift corresponds to a prefix sum or exclusive scan, and can be obtained via `MPI_Exscan` [Message Passing Interface Forum 2021].
Communication: This stage requires one global `MPI_Exscan` operation on a single integer of sufficient size to hold the largest DoF index.

At this stage, each process has (consecutively) enumerated a certain subset of DoFs, and we call these the *locally owned DoFs*. In later use, each process then owns the corresponding rows of matrices and entries in vectors, but the concept of locally owned DoFs is otherwise of no importance to the remainder of the algorithm. Importantly, however, we still need to ensure that each process learns of the remaining DoFs that are located on locally owned or ghost cells and whose indices are not currently known.

(5) *Ghost exchange*: In this step, we need to send sets of indices from those locally owned cells $K \in \mathbb{T}^p_{\text{loc}}$ that are ghost cells on other processes to those processes on which they are ghost cells. We do this in the following steps:
   (a) For each process $q \neq p$ that is adjacent to p, allocate a map with keys corresponding to global cell identifiers and values equal to a list of indices of those DoFs defined on this cell.
   (b) Iterate over all $K \in \mathbb{T}^p_{\text{loc}}$. If K is a ghost cell on process q, then add the global identifier of K and the list of DoFs on K to the map for process q.
   (c) Send all of the maps to their designed process q via nonblocking point-to-point communication (e.g., using `MPI_Isend` [Message Passing Interface Forum 2021]).
   (d) Receive data containers from processes of adjacent subdomains q via nonblocking point-to-point communication (e.g., using `MPI_Irecv` [Message Passing Interface Forum 2021]). The data so received corresponds to the DoF indices on all ghost cells of this subdomain p. On each of these cells, set the received DoF indices accordingly.

   All communication in this step is symmetric, which means that a process only receives data from another process when it also sends data to it. Thus, there is no need to negotiate communication.
   Communication: This stage requires point-to-point communication between all processes that own neighboring subdomains. The data sent consists of the indices of DoFs on all those cells that are locally owned by the sending process and that are ghost cells on the receiving process.

After this ghost exchange, each DoF on an interface between a locally owned and a ghost cell has exactly one valid index assigned.

(6) *Merge on interfaces*: Iterate over all locally relevant cells $K \in \mathbb{T}^p_{\text{rel}}$. On interfaces between locally owned and ghost cells, set all remaining invalid DoF indices to the corresponding valid one.
   Communication: This stage does not require any communication.

At this stage, all processes know the correct indices for all DoFs located on locally owned cells. However, during the ghost exchange in stage 5 above, some processes may have sent index sets

for some cells that may still contain the invalid index i and not all of these can be resolved through unification with locally known indices in stage 6. This is not illustrated in the figures but would require a larger example mesh; the source of these i markers are if a ghost cell owned by process q does not only border a cell owned by process p, but also a cell owned by yet another process q' that is not a neighbor of p, and process q will only learn about indices on this cell *as part of the ghost exchange with q' itself*. As a consequence, we have to repeat stage 5 one more time:

(7) *Ghost exchange (without illustration)*: Repeat the steps of stage 5. However, this time, only data from those cells have to be communicated that had invalid DoF indices prior to stage 5(d).
Communication: This stage requires point-to-point communication between all processes that own neighboring subdomains. The data sent consists of the indices of DoFs on a subset of all those cells that are locally owned by the sending process and that are ghost cells on the receiving process.

At the end of this algorithm, all global DoF indices have been set correctly, and every process knows the indices of DoFs located on locally owned and ghost cells (i.e., the "locally relevant DoFs" in the terminology of Bangerth et al. [2012]). The proof that this is so is given by the following three considerations. First, at the end of stage 4, we know the final indices of all DoFs on the locally owned cells with the exception of ones on interfaces to ghost cells if these DoFs are owned by another process. Second, in stage 5 of the algorithm, through the first ghost exchange, every process receives information about DoF indices from the owners of these DoFs. Stage 6, which merges DoFs on interfaces, then completes the knowledge of all DoFs on locally owned cells. Third, because now every process knows about all indices for the DoFs on locally owned cells, in stage 7, during the second ghost exchange, every process receives information from the owners of all ghost cells that allows completing all DoF indices on ghost cells.

Remark 1. In 3D scenarios, in Section 4.6 of the work of Bangerth and Kayser-Herold [2009], the authors point out possible complications with circular constraints during DoF unification whenever three or more different finite elements share a common edge. We have not found other satisfactory solutions for this problem in the intervening years and consequently continue to implement the suggestion in the work of Bangerth and Kayser-Herold [2009]: all DoFs on such edges are excluded from the unification step and will be treated separately via constraints. Since the decision to use or not use the unification algorithm on these edges is independent of whether the adjacent cells are on the same or different processes, this decision has no bearing on our overall goals of ensuring that the number of used indices be independent of the partitioning of the mesh. In the examples presented in Section 5, the fraction of identity constraints stays below 3%.

Remark 2. During stage 3 of our example in Figure 2, we follow the DoF unification procedure as described in the work of Bangerth and Kayser-Herold [2009]: if different finite elements meet on a subdomain interface, all shared DoFs will be assigned to the finite element representing the common function space (i.e., when using elements within the same family, the one with the lower polynomial degree). Of course, different decisions are possible, which might have an impact on parallel performance. For example, in Remark 2 in the work of Bangerth et al. [2012], the authors pointed out that on a face, all DoFs should belong to the same subdomain to speed up parallel matrix-vector multiplications. We implemented such an enumeration algorithm as an alternative to the one presented here. For the Laplace example used for the weighted load balancing experiments described in Section 5.2, we found that both implementations take the same runtime (<1% deviation).

3 LOAD BALANCING

To enable our algorithms to scale well, we need to ensure that each process does roughly the same amount of work. In contrast to *h*-adaptively refined meshes, a major difficulty here is that the workload per cell is not the same: different parts of the overall *hp*-adaptive algorithm scale differently with the number n_{DoFs} of unknowns per cell—for example, the cost of enumerating DoFs on a cell is proportional to n_{DoFs}, whereas assembling cell-local contributions to the global system costs $O(n_{\text{DoFs}}^3)$, unless one uses specific features of the finite element basis functions. More importantly, how the cost of a linear solver or **Algebraic Multigrid (AMG)** implementation—together the largest contribution to a program's runtime—scales with the polynomial degree or number of unknowns on a cell is quite difficult to estimate *a priori*. As a consequence, when using different polynomial degrees on different cells, it is not easy to derive theoretically what the computational *cost* of a cell is going to be, and consequently how to weigh each cell.

Oden et al. [1994] and Patra and Oden [1995] investigate different decomposition and load balancing strategies with various types of weights, which are closely tied to their *hp*-adaptive algorithm. These studies use the number of DoFs on a given cell as a natural choice for the weight of that cell, but we believe that this does not reflect the computational effort accurately for the reasons pointed out earlier. Herein, we instead use an empirical approach in which we assume that the relative cost w of a cell K can be expressed as

$$w(K) = n_{\text{DoFs}}(K)^c, \tag{1}$$

with some, *a priori* unknown, exponent c. During load balancing, we then weigh each cell with this factor and seek to partition meshes so that the sums of weights of the cells in each partition are roughly equal. For this purpose, we use the partitioning algorithms provided by the `p4est` library that are described in Section 3.3 of the work of Burstedde et al. [2011].

We experimentally determine the value for the exponent c for which the overall runtime of our program is minimized and show results to this end in Section 5.2. From the preceding considerations, one would expect that the minimum should be in the range $1 \leq c \leq 3$, and this indeed turns out to be the case.

It is worth mentioning that the approach only minimizes the *overall* runtime but likely leaves each individual operation suboptimally load balanced. This imbalance is a common problem when a program executes algorithms whose cell-local costs are not proportional (e.g., see the work of Gassmöller et al. [2018]) and can only be solved by re-partitioning data structures between the different phases of a program—say, between matrix assembly and the actual solver phase. Exploring this issue is beyond the scope of our study; moreover, as we will show in Section 5, for the test cases we consider, the solver's contribution to the overall runtime is so dominant that it is not worth trying to better load balance any of the nonsolver components.

4 PACKING, UNPACKING, AND TRANSFERRING DATA

A frequent operation in finite element codes is the serialization of all information associated with a cell into an array, and moving this data. Examples for where this operation is relevant are repartitioning a mesh among processes after refinement and the generation of checkpoints for later restart. In such cases, it is often convenient to write all information associated with the cells of one process into contiguous buffers.

For *h*-adaptive meshes, this presents few challenges since the size of the data associated with every cell is the same and, consequently, can be packed into buffers of fixed size per cell. However, for *hp*-methods, different cells require different buffer sizes for efficiency, and creating contiguous storage schemes for all data on each process requires a bit more thought. Thought is also necessary when devising mechanisms to subsequently transfer this data to other processes.

In practice, we implement such schemes using a two-stage process. In a first stage, we assess how much memory the data on each cell requires and allocate a contiguous array that can hold information from all cells. In this phase, we also build a second array that holds the offsets into the first array at which the data from each cell starts. The storage scheme therefore resembles the way sparse matrices are commonly stored in compressed row storage. In a second stage, we copy the actual data from each cell into the respective part of the array.

For serialization, one can then write the two arrays in their entirety to disk. For re-partitioning, parts of the arrays have to be sent to different processes based on which process will own a cell. For this step, it is useful to sort the order in which cells are represented in the two arrays in such a way that data destined for one target is stored as one contiguous part of the arrays. In this way, all information to be sent to one process can be transferred with a single nonblocking point-to-point send operation (with nonuniform buffer sizes) for each of the two arrays, without the need for further copy operations. How this can be efficiently implemented is described in detail in Section 5.2 of the work of Burstedde [2020].

5 NUMERICAL RESULTS

Ultimately, the algorithms we presented in Sections 2, 3, and 4 are only useful if they can be efficiently implemented. In this section, we assess our approaches using two test cases: a 2D Laplace problem and a 3D Stokes problem. We discuss these in Section 5.1.

Based on these test problems, we first assess how one needs to choose load balancing weights for each cell based on the polynomial degree of the finite element applied (Section 5.2). Using the resulting load balancing strategy, we then discuss how our algorithms scale in Section 5.3; an important question to discuss in this context will be how one would actually define and measure "scalability" in the context of *hp*-adaptive methods.

All of the results shown in this section have been obtained using codes that are variations of tutorial programs of the `deal.II` library. All features discussed in this article are implemented in `deal.II` (also see the work of Arndt et al. [2021, 2022]). All data were generated with the tool `hpbox` [Fehling 2022].

5.1 Test Cases

We evaluate the performance of our algorithms using two test cases discussed next: a 2D Laplace equation posed on the L-shaped domain and a 3D Stokes problem posed on a domain that resembles a forked ("Y"-shaped) pipe. Both of these cases are chosen because the domain induces corner singularities in the solution, resulting in parts of the domain where either large cells with high-order elements or small cells with low-order elements are best suited to approximate the exact solution. In other words, these cases mimic practical situations that are well suited to *hp*-adaptive methods. Furthermore, being able to demonstrate our algorithms on both a relatively simple, scalar 2D problem and a much more complex 3D, coupled vector-valued problem illustrates the range and limitations of our algorithms.

In each test case, we start from a coarse discretization of the problem, solve it, and refine it in multiple iterations to end up with a mesh tailored to the problem. For this purpose, we need mechanisms to decide which cells we want to refine and how. We use an error estimator based on the work of Kelly et al. [1983] to mark cells for general refinement. Further, we use a smoothness estimator based on the decay of Legendre coefficients as described in other works [Eibner and Melenk 2007; Houston and Süli 2005; Mavriplis 1994] to decide how we want to refine each cell. We employ fixed number refinement for both *h*- and *p*-refinement, which means that the fraction of cells we are going to refine is always the same. We state our choice of fractions in the descriptions that follow. The mesh is repartitioned after each refinement iteration.

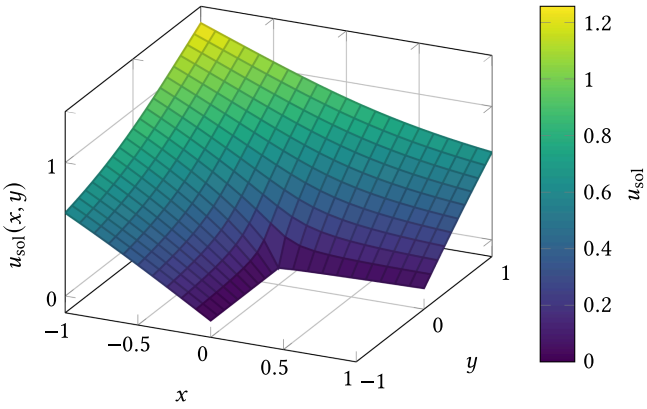


Fig. 3. The solution (3) of the Laplace problem (2) on the L-shaped domain.

5.1.1 Test Case 1: A Laplace Problem on the L-Shaped Domain. Our first test case concerns the solution of the Laplace problem with Dirichlet boundary conditions:

$$-\Delta u(x) = 0 \quad \text{on} \quad \Omega\,, \qquad\qquad u(x) = u_{\text{sol}}(x) \quad \text{on} \quad \partial\Omega, \tag{2}$$

where we choose $\Omega \subset \mathbb{R}^2$ as the L-shaped domain, $\Omega = (-1, 1)^2 \backslash [0, 1]\times[-1, 0]$. It is well understood that on such domains, the Laplace equation admits a singular solution; indeed, in polar coordinates $r = \sqrt{x^2 + y^2} > 0$ and $\theta = \arctan(y/x)$, the function

$$u_{\text{sol}}(x) = r^\alpha \sin(\alpha\, \theta) \tag{3}$$

is a solution for an opening angle at the re-entrant corner of π/α with $\alpha \in (1/2, 1)$. For the L-shaped domain, we have $\alpha = 2/3$ and the corresponding solution is shown in Figure 3. We impose $u = u_{\text{sol}}$ as the boundary condition on $\partial\Omega$, and the resulting (exact) solution of the Laplace equation that we seek to compute is then $u = u_{\text{sol}}$ everywhere in Ω.

This solution is singular at the origin: with unit vectors $e_r = \cos(\theta)e_x + \sin(\theta)e_y$ and $e_\theta = -\sin(\theta)e_x + \cos(\theta)e_y$, we find that

$$\nabla u_{\text{sol}}(x) = \alpha r^{\alpha-1}\,[\sin(\alpha\, \theta)e_r + \cos(\alpha\, \theta)e_\theta]\,, \tag{4}$$

and consequently $\lim_{r\to 0} \|\nabla u_{\text{sol}}(x)\|_2 = \infty$ for our choice of α.

The numerical solution of the Laplace equation on the L-shaped domain is a classical test case. For example, Mitchell and McClain [2014] present several benchmarks for *hp*-adaptation for this situation. A similar scenario is also used in the step-75 tutorial of the `deal.II` library [Fehling et al. 2021].

In our study, we choose Lagrange elements Q_k with polynomial degrees $k = 2, \ldots, 7$. We mark 30% of cells for refinement and 3% for coarsening, from which we pick 90% to be *p*-adapted and 10% to be *h*-adapted. We choose to favor *p*-refinement since the only nonsmooth part of the solution is around the point singularity at the origin.

Figure 4 shows a typical *hp*-mesh and its partitioning from a sequence of adaptive refinements. It illustrates that the corner singularity requires *h*-adaptation resulting in small cells, whereas farther away from the corner, the solution is smoother and can be resolved on relatively coarse meshes using high polynomial degrees. Far away from the origin, the estimated errors are low so that large cells and low polynomial degrees are sufficient. The lobe pattern results from the anisotropic resolution property of polynomials on quadrilaterals.

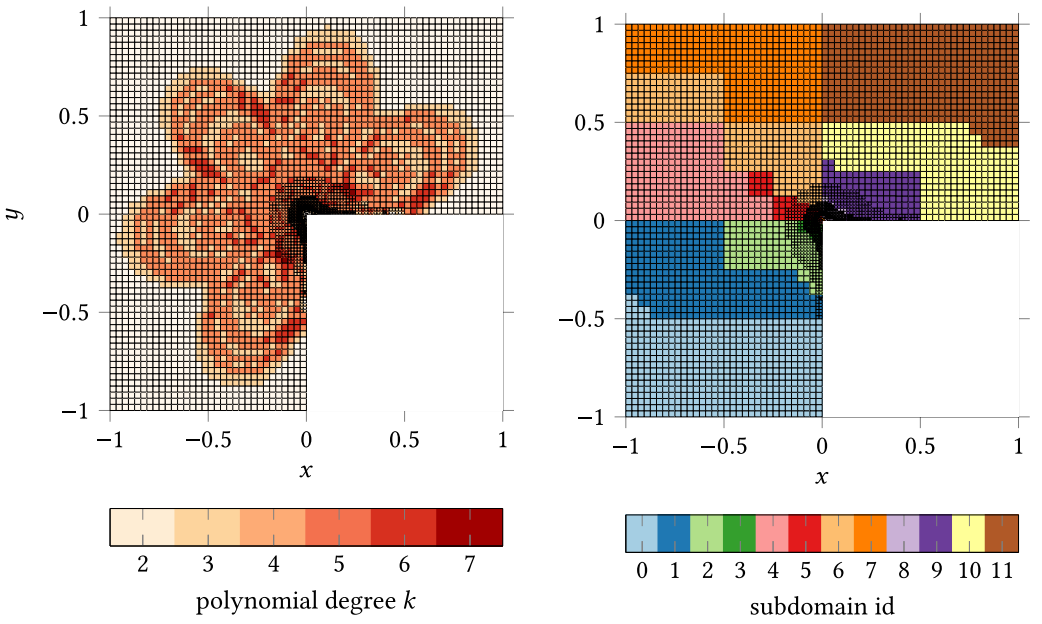


Fig. 4. Numerical approximation of the Laplace problem (2) after six adaptation cycles and five initial global refinements. Left: The mesh and polynomial degrees used on each cell. Right: Partitioning of the mesh onto 12 MPI processes with a load balancing weighting exponent of $c = 1.9$.

The numerical scheme we choose to solve this problem is based on `Trilinos` [Heroux et al. 2005] for parallel linear algebra and uses the `ML` package [Gee et al. 2007] as an AMG preconditioner inside a conjugate gradient iteration.

5.1.2 Test Case 2: Flow Through a Y-Pipe. As a second test case, we consider the solution of the Stokes equation describing slow flow,

$$-\Delta u + \nabla p = \mathbf{0}, \tag{5a}$$

$$-\nabla \cdot u = 0. \tag{5b}$$

As the domain, we choose a forked, "Y"-shaped pipe (Figure 5). We impose no-slip boundary conditions on the lateral surfaces ($u = 0$) and model the inflow at one opening as a Poisseuille flow via Dirichlet boundary conditions. The other two ends are modeled via zero-traction boundary conditions. Velocity and pressure solutions are also shown in Figure 5.

The "welding seams" at which the three pipes meet are nonconvex parts of the boundary, again resulting in singular solutions where we expect that the gradient of the velocity u becomes infinite; the pressure is also singular at these locations. We chose this as the second test case because it enables us to verify that enumerating DoFs, along with all of the other ingredients of our *hp*-adaptive solution approach, are efficient and scale well also for 3D problems with the much more complex choice of finite element and solver techniques necessary to solve the Stokes problem.

In particular, we use "Taylor-Hood" type elements $\boldsymbol{Q}_k/Q_{k-1}$ [Taylor and Hood 1973], where the three components of the velocity solution use elements of polynomial degree k and the single component of the pressure uses an element of polynomial degree $(k-1)$. In our study, we choose a collection of elements with $k = 3, \ldots, 6$.

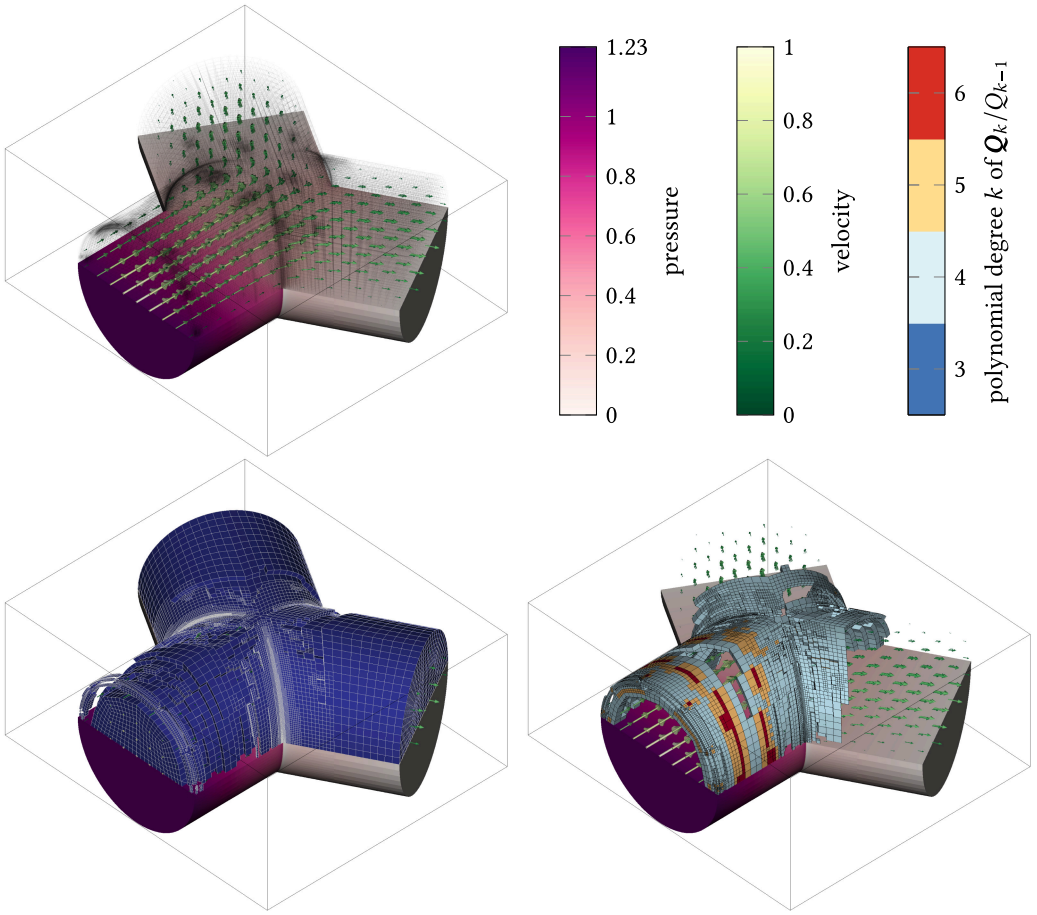


Fig. 5. Stokes flow through the Y-pipe as described by Equation (5) after four adaptation cycles and three initial refinements. Top: The lower half shows the domain and the pressure, whereas the upper half depicts the mesh and a vector plot describing the velocity field. Bottom left: A cut-away showing those cells with low polynomial degrees, located generally where either the estimated errors are low or near the nonconvex parts of the domain. Bottom right: Cut-away showing those cells with a high polynomial degree.

Both refinement and *hp*-decision indicators are based on the scalar-valued pressure solution. We mark 10% of cells for refinement and 1% for coarsening, which we divide equally into being *h*- and *p*-adapted.

We solve the linear saddle point system that results from discretization using flexible GMRES [Saad 1993] and a Silvester-Wathen-type preconditioner [Silvester and Wathen 1994] in which we treat the elliptic block with the `ML` algebraic multigrid (AMG) preconditioner [Gee et al. 2007] of `Trilinos` [Heroux et al. 2005]. This combination of solver and preconditioner is known to scale to very large problems, at least for elements of fixed order (see the work of Bangerth et al. [2012] and Kronbichler et al. [2012]).

Remark 3. In our experiments, we have found that the AMG solver used in both of the test cases struggles with increasing fragmentation of polynomial degrees in the mesh. To address this, we limit the difference of polynomial degrees on neighboring cells to one, in a scheme not dissimilar

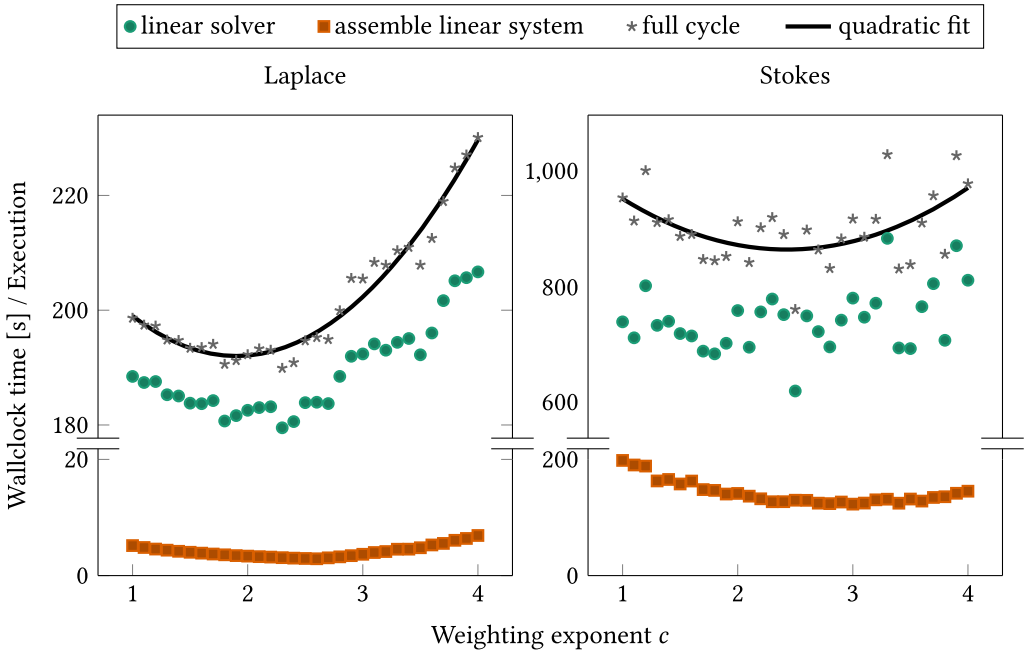


Fig. 6. Wallclock times for several operations of a complete adaptation cycle, when partitioning the mesh using different weighting exponents c (see (1)). Left: For one cycle of the 2D Laplace problem of Section 5.1.1. The problem has about 51 million DoFs and is solved on 96 MPI processes. Right: For one cycle of the 3D Stokes problem of Section 5.1.2. The problem has about 15 million DoFs and is solved on 96 MPI processes.

to the commonly used approach of only allowing neighboring cells to differ by at most one level in mesh refinement. In our experiments, this "smoothing" of polynomial degrees reduces the number of solver iterations by up to 70%; this translates equally to the wallclock time spent on solving the linear system.

5.2 Load Balancing

As mentioned in Section 3, it is not clear *a priori* how to weigh the contribution of each cell of a mesh to the overall cost of a program. As a consequence—and unlike the h-adaptive case—it is not clear what the optimal load balancing strategy is.

Using the weighting proposed in Section 3, we have therefore run numerical experiments that vary the relative weighting of cells based on the number of DoFs on each cell. We carry out investigations on a mesh with a wide variety of polynomial degrees and a substantial number of hanging nodes that we obtain after a number of mesh refinement cycles. We keep this particular mesh but partition it differently onto the available MPI processes for varying values of the weighting exponent c in (1), and run a complete refinement cycle involving enumeration of DoFs, assembly of the linear system, and solution of the linear system on the so-partitioned mesh. These experiments were run on a workstation with two AMD EPYC 7552 processors (with 48 cores each, running at 2.2 GHz) and 512 GB of memory.

The results are shown in Figure 6 for the two test cases defined in Section 5.1. For both cases, the largest contribution to the overall cost is the linear solver; the noise in the corresponding curves results from slightly different numbers of linear solver iterations, likely a consequence of decisions made in how the AMG algorithm builds its hierarchy in response to which rows of the

overall matrix are stored on which process. The data shown in Figure 6 suggest that the overall runtime is minimized with an exponent of $c \approx 1.9$ for the Laplace test case and $c \approx 2.4$ for the Stokes test case. We use these values for the weighting exponent for all other experiments shown in the following.

The data shown in the figure makes it clear that the optimal exponent depends on the problem solved and needs to be assessed for each problem individually. However, in general, the dependency of the runtime on the specific choice of exponent is relatively weak.

5.3 Efficiency and Scalability of Algorithms

Next, we assess whether the algorithms we proposed in Section 2 are efficient and scale to large problem sizes. To answer this question, we first discuss what "scalability" means in the context of *hp*-adaptive methods, before turning to results obtained on the test cases defined in Section 5.1.

All results shown in this section were obtained on the Expanse supercomputer. Each standard computing node is equipped with two AMD EPYC 7742 processors (with 64 cores each, running at 2.25 GHz) and 256 GB of DDR4 DRAM memory. Communication between nodes happens via a Mellanox HDR-100 InfiniBand Interconnect network operating in a hybrid fat-tree topology. More information on the configuration of the machine can be found in the work of Strande et al. [2021].

5.3.1 How to Define Scalability? One typically measures the efficiency of a parallel algorithm running on P processes operating in parallel on N work items through either "strong scaling" (where the problem size N is fixed and we vary the number of processes P) or "weak scaling" (where one increases the problem size N along with the number of processes P, keeping N/P constant). In both cases, one measures the time it takes the algorithm to complete work.

For *h*-adaptive algorithms, it is relatively straightforward to define what N is supposed to be: it could be (i) the number of cells in the mesh, (ii) the number of unknowns in a finite element discretization on that mesh (which equals the size of the linear systems that result), or (iii) the number of nonzero entries in the matrix (which determines the cost of a matrix-vector product but is also an important consideration in the cost of algorithms like AMG). The choice of which of these we want to call N is unimportant *because they are all proportional to each other.* Indeed, if one uses an optimal solver such as multigrid, one could also (iv) define N to be the number of floating point operations required to solve the linear system for a given problem—it is again proportional to the other measures.

But things are not this easy for *hp*-adaptive methods: when using different polynomial degrees on cells, the four quantities mentioned earlier are no longer proportional to each other when considering an *hp*-fragmented mesh. This disproportionality is of no importance when considering strong scalability, because the problem size N is fixed. But it is not obvious how to define weak scalability because a sequence of problems that keeps N/P constant for one definition of N may not imply that N/P is constant for any of the other definitions of N. Similarly, we show results in the following where we increase N for fixed P, observing how time scales with N—for which, again, the observed scaling depends on what definition of N we choose.

As a consequence, we describe results in the following where we either use $N = N_{\text{DoFs}}$ (the number of global DoFs in the problem) or $N = N_{\text{nonzeros}}$ (the global number of nonzero entries in the matrix that needs to be solved with on a given mesh). As expected, we will see that operations such as the assembly of a linear system and its solution do not scale with the problem size as $O(N_{\text{DoFs}})$, but they instead scale close to the amount of work as $O(N_{\text{nonzeros}})$.

5.3.2 Results for the Laplace Test Case of Section 5.1.1. With these considerations in mind, let us now turn to concrete timing data. Next, we show results for how much time our implementation

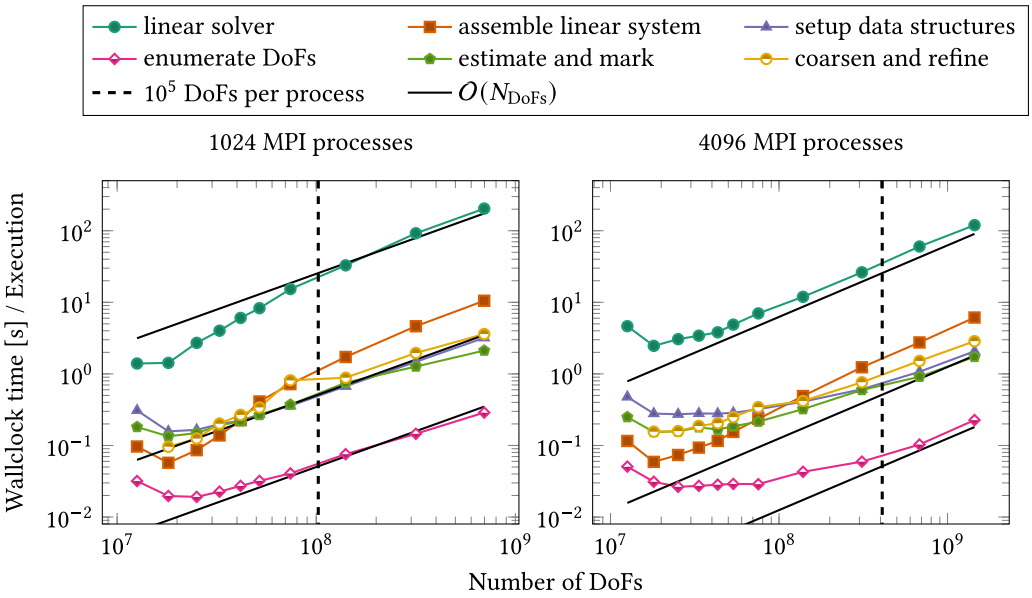


Fig. 7. Laplace problem. Scaling of wallclock time as a function of the number of unknowns N_{DoFs} on a sequence of consecutively refined meshes, for 1024 (left) and 4096 MPI processes. Each MPI process owns more than 10^5 DoFs only to the right of the indicated vertical line; to the left of this line, processes do not have enough work to offset the cost of communication, and parallel efficiency should not be expected. The solid black trend lines indicate optimal scaling, $O(N_{\text{DoFs}})$. Since the computations on the right were done on four times as many processes as on the left, we have offset the black trend lines downward by a factor of 4: assuming optimal strong scaling when increasing the number of MPI processes, a point located on the trend line in the left subfigure should have a corresponding point on the offset trend line also in the right subfigure.

of the Laplace test case of Section 5.1.1 spends in each of the following categories of operations (ordered roughly in their relevance to the overall runtime to the program):

- *Linear solver*: This category includes setting up the AMG preconditioner and then solving the linear system.
- *Assemble linear system*: Compute cell-local matrix and right-hand-side vector contributions to the linear system, and insertion into the global objects. This step also includes communicating these contributions to the process owning a matrix or vector row if necessary.
- *Setup data structures*: This step includes a number of setup steps that happen after generating a mesh and before the assembly of the linear system. Specifically, we include the enumeration of DoFs, exchanging between processes which non-locally owned matrix entries they will write into, setting up a sparsity pattern for the global matrix, allocation of memory for the system matrix and vectors, and determining constraints that result from hanging nodes and boundary conditions.
- *Enumerate DoFs*: This category, a subset of the previous one, measures the time to enumerate all DoFs based on the algorithm discussed in Section 2.3.
- *Estimate and mark*: Once the linear system has been solved, this step computes error and smoothness estimates for each locally owned cell. It then computes global thresholds for *hp*-adaptation and flags cells for either *h*- or *p*-adaptation.
- *Coarsen and refine*: This final step performs the actual *h*-adaptation on marked cells while enforcing a 2:1 cell size relationship across faces. It also updates the associated finite element

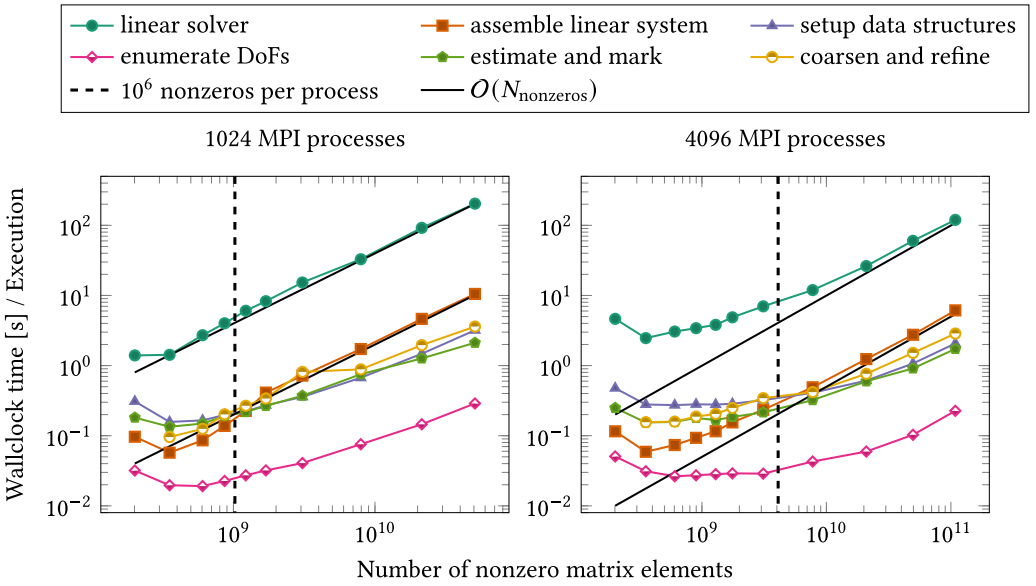


Fig. 8. Laplace problem. The same scaling data as shown in Figure 7, except shown as a function of the nonzero entries of the matrix.

on cells (*p*-adaptation) while limiting the difference of polynomial degrees across cell interfaces. This category also measures the transfer data between old and new mesh, as well as the cost of re-partitioning the mesh between processes.

Figure 7 shows timing information for a situation where we repeatedly solve the problem while adaptively refining the *hp*-mesh, on both 1024 and 4096 MPI processes. In this setup, with a fixed number P of processes, one would hope that the runtime increases linearly with the problem size N. Our results demonstrate that this linearity holds when N is the N_{DoFs} on each of the meshes—at least once the problem is large enough. Importantly for the current work, operations such as estimating *hp*-indicators and refining the mesh accordingly, and particularly the enumeration of DoFs using the algorithm of Section 2.3, are only minor contributions to the overall runtime, which is dominated by the assembly and particularly the solution of linear systems.

However, Figure 7 also shows that both the assembly and the solution of the linear system do not scale like $O(N_{\text{DoFs}})$. This result may not be surprising in view of the discussions of Section 5.3.1: as we move from left to right, we not only increase the number of unknowns but also increase polynomial degrees on cells, resulting in denser and denser linear systems that are more costly to assemble and solve. As a consequence, Figure 8 shows the same data as a function of the nonzero entries N_{nonzeros}. This figure illustrates that using this definition, both assembly and the solution of linear systems scale nearly perfectly as $O(N_{\text{nonzeros}})$.

A comparison of the left and right panels of Figures 7 and 8—and particularly how the various curves approach the trend lines $O(N)$ that are offset in the panels by the ratio of the number of MPI processes used—shows that for sufficiently large problems, we also have good *strong* scaling. We expand on this in Figure 9, where we show scaling for a fixed problem with the number of processes. The figure shows that most operations may not scale perfectly as $O(1/P)$, but scalability is at least adequate as long as the problem size per process remains sufficiently large (to the left of the dashed line). The exception is the performance of the linear solver; this is a known problem with implementations of AMG methods but also beyond the scope of the current work.

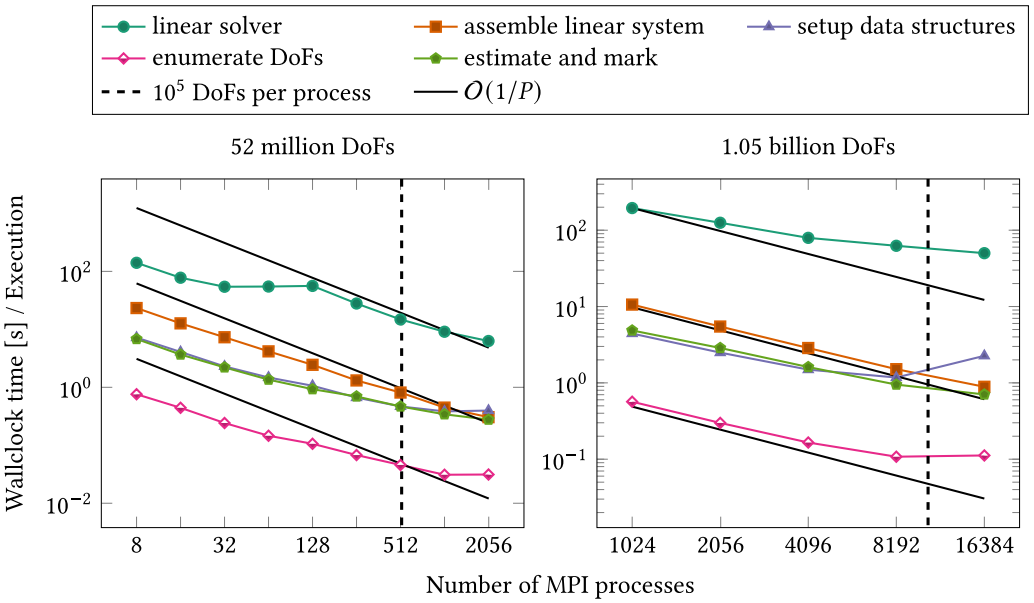


Fig. 9. Laplace problem. Strong scaling for one advanced adaptation cycle at different problem sizes. Each MPI process owns more than 10^5 DoFs only to the left of the indicated vertical line; to the right of this line, processes do not have enough work to offset the cost of communication, and parallel efficiency should not be expected. Left: Fixed problem size of roughly 52 million DoFs. Right: Fixed problem size of roughly 1.05 billion DoFs. The trend lines for $O(1/P)$ are offset between the two panels by the ratio of the size of the problem to allow for assessing weak scalability of the algorithms.

5.3.3 Results for the Stokes Test Case of Section 5.1.2. We repeat many of these timing studies, using the same timing categories, for the Stokes test case to assess whether our results also hold for a more complex, 3D, and vector-valued problem.

The left panel of Figure 10 illustrates how runtime scales with the size of the problem (here measured by the number of global DoFs N_{DoFs}) and again shows that most operations scale as one would expect given the results of the previous section.

The right panel of Figure 10 presents strong scaling data. As before, we get good strong scalability as long as the problem size per process is sufficiently large (to the left of the dashed line). At the same time, the figure also illustrates the limitations imposed by the linear solver we use and that have prevented us from considering larger problems: much larger problems would have taken many hours to solve even with large numbers of processes. We did not believe that the associated expense in CPU cycles would have provided further insight that is not already clear from the results of the previous section and the figure—namely, that with the exception of the linear solver and possibly assembly, all *hp*-related operations scale reasonably well to large problem sizes for both simple (2D Laplace) and complex (3D Stokes) problems.

6 CONCLUSION

In this article, we have presented algorithms combining our previous work on parallel and *hp*-adaptive FEMs, and that allows us to solve problems with *hp*-adaptive methods on large, parallel machines with distributed memory. In particular, we have presented algorithms for the enumeration of DoFs, a heuristic approach to weighted load balancing, and on how to transfer data of variable size between processes.

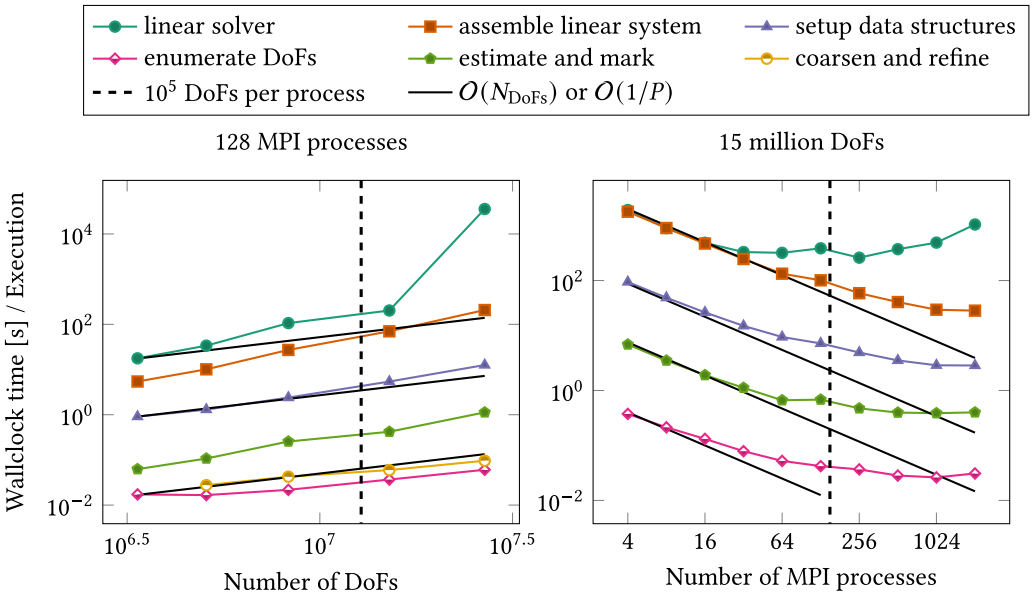


Fig. 10. Stokes problem. Left: Consecutive adaptation cycles with 128 MPI processes. The dashed line again indicates 10^5 DoFs per process; processes have more than this number only to the right of the line. Right: Strong scaling with a fixed problem of 15 million DoFs. Computations exceed 10^5 DoFs per process only to the left of the dashed line.

The results we have shown in Section 5 illustrate that our algorithms all scale reasonably well both to large problems and large MPI process counts, and particularly—as one might have expected—that (i) the linear solver is the bottleneck in solving partial differential equations that result from *hp*-discretizations, and that (ii) the enumeration algorithm of Section 2 contributes to the overall runtime in an essentially negligible way.

Although the second of these conclusions makes clear that we have succeeded in our algorithm design goals—we have come up with an algorithm for a task for which there was none before, and the algorithm is fast enough to not be a bottleneck—our data also clearly points to future work necessary to make *hp*-methods viable for more widespread use: we need more scalable iterative solvers and preconditioners, specifically ones that are better than the AMG ones we have used here. Such work would, for example, build on the geometric multigrid ideas in the work of Jomo et al. [2021] and Mitchell [2010] or hybrid approaches like those of Brown et al. [2022] and Fehn et al. [2020]. Furthermore, the literature suggests that the matrix-free approaches of Brown et al. [2022], Kronbichler and Kormann [2012], and Munch et al. [2022] should be able to overcome many of these solver limitations.

ACKNOWLEDGMENTS

This article is dedicated to the memory of William (Bill) F. Mitchell (1955–2019), who for many years moved the *hp*-finite element method along by providing high-quality implementations of the method through his PHAML software [Mitchell 2002] when there were few other packages that one could play with. Equally importantly, in a monumental effort, he collected and compared the many different ways proposed in the literature in which one can drive *hp*-adaptivity in practice. This work—an extension of his work in the late 1980s comparing *h*-adaptive refinement criteria [Mitchell 1989]—resulted in a comprehensive 2014 paper that in the end stood at 39 pages [Mitchell

and McClain 2014], but for which the original 2011 NIST report had a full 215 pages [Mitchell and McClain 2011]. Other significant contributions of Bill are in the area of parallelization and load balancing in form of the REFTREE algorithm [Mitchell 2007].

Computational methods only gain broad acceptance when the literature contains incontrovertible evidence in the form of comparison *between* methods. Papers that do such comparisons are tedious to write and often not as highly regarded as ones that propose new methods, but crucial for our community to finally see which methods work and which don't. Bill excelled at writing such papers, and his contributions to *hp*-finite element methods will continue to be highly regarded. His impartial and objective approach to declaring winners and losers will be missed!

An obituary for Bill Mitchell can be found in the work of Boisvert [2019].

REFERENCES

Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. 2015. The FEniCS project version 1.5. *Archive of Numerical Software* 3, 100 (Dec. 2015), 9–23. https://doi.org/10.11588/ans.2015.100.20553

Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, et al. 2021. MFEM: A modular finite element methods library. *Computers & Mathematics with Applications* 81 (Jan. 2021), 42–74. https://doi.org/10.1016/j.camwa.2020.06.009

Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, and David Wells. 2021. The deal.II finite element library: Design, features, and insights. *Computers & Mathematics with Applications* 81 (2021), 407–422. https://doi.org/10.1016/j.camwa.2020.02.022

Daniel Arndt, Wolfgang Bangerth, Marco Feder, Marc Fehling, Rene Gassmöller, Timo Heister, Luca Heltai, et al. 2022. The deal.II library, version 9.4. *Journal of Numerical Mathematics* 30, 3 (July 2022), 231–246. https://doi.org/10.1515/jnma-2022-0054

Ivo Babuška and Milo R. Dorr. 1981. Error estimates for the combined h and p versions of the finite element method. *Numerische Mathematik* 37, 2 (June 1981), 257–277. https://doi.org/10.1007/bf01398256

Ivo Babuška and Benqi Guo. 1996. Approximation properties of the h-p version of the finite element method. *Computer Methods in Applied Mechanics and Engineering* 133, 3-4 (July 1996), 319–346. https://doi.org/10.1016/0045-7825(95)00946-9

Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. 2012. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software* 38, 2 (Jan. 2012), Article 14, 28 pages. https://doi.org/10.1145/2049673.2049678

Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. 2007. deal.II - A general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software* 33, 4 (Aug. 2007), Article 24, 27 pages. https://doi.org/10.1145/1268776.1268779

Wolfgang Bangerth and Oliver Kayser-Herold. 2009. Data structures and requirements for *hp* finite element software. *ACM Transactions on Mathematical Software* 36, 1 (March 2009), Article 4, 31 pages. https://doi.org/10.1145/1486525.1486529

Peter Bastian, Markus Blatt, Andreas Dedner, Nils-Arne Dreier, Christian Engwer, René Fritze, Carsten Gräser, et al. 2021. The DUNE framework: Basic concepts and recent developments. *Computers & Mathematics with Applications* 81 (Jan. 2021), 75–112. https://doi.org/10.1016/j.camwa.2020.06.007

Kim S. Bey, Abani Patra, and John Tinsley Oden. 1996. *hp*-Version discontinuous Galerkin methods for hyperbolic conservation laws. *Computer Methods in Applied Mechanics and Engineering* 133, 3 (July 1996), 259–286. https://doi.org/10.1016/0045-7825(95)00944-2

Ronald F. Boisvert. 2019. A Tribute to William F. Mitchell. Retrieved June 8, 2023 from https://sinews.siam.org/Details-Page/a-tribute-to-william-f-mitchell.

Susanne Brenner and Ridgway Scott. 2008. *The Mathematical Theory of Finite Element Methods* (3rd ed.). Springer, New York, NY. https://doi.org/10.1007/978-0-387-75934-0

Jed Brown, Valeria Barra, Natalie Beams, Leila Ghaffari, Matthew Knepley, William Moses, Rezgar Shakeri, Karen Stengel, Jeremy L. Thompson, and Junchao Zhang. 2022. Performance portable solid mechanics via matrix-free p-multigrid. *arXiv:2204.1722 [cs, math]* (2022). https://doi.org/10.48550/arXiv.2204.01722

Carsten Burstedde. 2020. Parallel tree algorithms for AMR and non-standard data access. *ACM Transactions on Mathematical Software* 46, 4 (Nov. 2020), Article 32, 31 pages. https://doi.org/10.1145/3401990

Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. 2011. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing* 33, 3 (2011), 1103–1133. https://doi.org/10.1137/100791634

Noel Chalmers, Gbemeho Agbaglah, Marcin Chrust, and Catherine Mavriplis. 2019. A parallel *hp*-adaptive high order discontinuous Galerkin method for the incompressible Navier-S equations. *Journal of Computational Physics: X* 2 (March 2019), Article 100023, 22 pages. https://doi.org/10.1016/j.jcpx.2019.100023

Tino Eibner and Jens Markus Melenk. 2007. An adaptive strategy for *hp*-FEM based on testing for analyticity. *Computational Mechanics* 39, 5 (April 2007), 575–595. https://doi.org/10.1007/s00466-006-0107-0

Marc Fehling. 2020. *Algorithms for Massively Parallel Generic hp-Adaptive Finite Element Methods*. Ph. D. Dissertation. Bergische Universität Wuppertal, Forschungszentrum Jülich GmbH. http://hdl.handle.net/2128/25427

Marc Fehling. 2022. hpbox: Sandbox for *hp*-adaptive methods. *Zenodo*. Retrieved June 8, 2023 from https://zenodo.org/record/6245947.

Marc Fehling, Peter Munch, and Wolfgang Bangerth. 2021. The deal.II tutorial step-75: Parallel *hp*-adaptive multigrid methods for the Laplace equation. *Zenodo*. Retrieved June 8, 2023 from https://zenodo.org/record/7741470.

Niklas Fehn, Peter Munch, Wolfgang A. Wall, and Martin Kronbichler. 2020. Hybrid multigrid methods for high-order discontinuous Galerkin discretizations. *Journal of Computational Physics* 415 (Aug. 2020), 109538. https://doi.org/10.1016/j.jcp.2020.109538

Rene Gassmöller, Harsha Lokavarapu, Eric Heien, Elbridge Gerry Puckett, and Wolfgang Bangerth. 2018. Flexible and scalable particle-in-cell methods with adaptive mesh refinement for geodynamic computations. *Geochemistry, Geophysics, Geosystems* 19, 9 (2018), 3596–3604. https://doi.org/10.1029/2018GC007508

Michael W. Gee, Christopher M. Siefert, Jonathan J. Hu, Ray S. Tuminaro, and Marzio G. Sala. 2007. *ML 5.0 Smoothed Aggregation User's Guide*. Technical Report SAND2006-2649. Sandia National Laboratories.

Christoph Gersbacher. 2016. Implementation of *hp*-adaptive discontinuous finite element methods in DUNE-FEM. *arXiv:1604.07242 [cs]* (2016). https://doi.org/10.48550/arXiv.1604.07242

Benqi Guo and Ivo Babuška. 1986a. The *h-p* version of the finite element method, part 1: The basic approximation results. *Computational Mechanics* 1, 1 (March 1986), 21–41. https://doi.org/10.1007/BF00298636

Benqi Guo and Ivo Babuška. 1986b. The *h-p* version of the finite element method, part 2: General results and applications. *Computational Mechanics* 1, 3 (Sept. 1986), 203–220. https://doi.org/10.1007/bf00272624

Frédéric Hecht. 2012. New development in freefem++. *Journal of Numerical Mathematics* 20, 3-4 (Dec. 2012), 251–266. https://doi.org/10.1515/jnum-2012-0013

Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, et al. 2005. An overview of the Trilinos project. *ACM Transactions on Mathematical Software* 31, 3 (Sept. 2005), 397–423. https://doi.org/10.1145/1089014.1089021

Paul Houston and Endre Süli. 2005. A note on the design of *hp*-adaptive finite element methods for elliptic partial differential equations. *Computer Methods in Applied Mechanics and Engineering* 194, 2-5 (Feb. 2005), 229–243. https://doi.org/10.1016/j.cma.2004.04.009

John Njuguna Jomo. 2021. *Towards Scalable Finite Cell Computations on Massively Parallel Systems*. Ph. D. Dissertation. Technische Universität München. https://mediatum.ub.tum.de/?id=1576808.

John Njuguna Jomo, Oguz Oztoprak, Frits de Prenter, Nils Zander, Stefan Kollmannsberger, and Ernst Rank. 2021. Hierarchical multigrid approaches for the finite cell method on uniform and multi-level hp-refined grids. *Computer Methods in Applied Mechanics and Engineering* 386 (Dec. 2021), 114075. https://doi.org/10.1016/j.cma.2021.114075

John Njuguna Jomo, Nils Zander, Mohamed Elhaddad, Ali Özcan, Stefan Kollmannsberger, Ralf-Peter Mundani, and Ernst Rank. 2017. Parallelization of the multi-level *hp*-adaptive finite cell method. *Computers & Mathematics with Applications* 74, 1 (July 2017), 126–142. https://doi.org/10.1016/j.camwa.2017.01.004

Łukasz Kaczmarczyk, Zahur Ullah, Karol Lewandowski, Xuan Meng, Xiao-Yi Zhou, Ignatios Athanasiadis, Hoang Nguyen, et al. 2020. MoFEM: An open source, parallel finite element library. *Journal of Open Source Software* 5, 45 (Jan. 2020), Article 1441, 8 pages. https://doi.org/10.21105/joss.01441

D. W. Kelly, J. P. De S. R. Gago, O. C. Zienkiewicz, and I. Babuska. 1983. *A posteriori* error analysis and adaptive processes in the finite element method: Part I—Error analysis. *International Journal for Numerical Methods in Engineering* 19, 11 (1983), 1593–1619. https://doi.org/10.1002/nme.1620191103

Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey. 2006. `libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers* 22, 3 (Dec. 2006), 237–254. https://doi.org/10.1007/s00366-006-0049-3

Martin Kronbichler, Timo Heister, and Wolfgang Bangerth. 2012. High accuracy mantle convection simulation through modern numerical methods. *Geophysical Journal International* 191, 1 (Oct. 2012), 12–29. https://doi.org/10.1111/j.1365-246X.2012.05609.x

Martin Kronbichler and Katharina Kormann. 2012. A generic interface for parallel cell-based finite element operator application. *Computers & Fluids* 63 (June 2012), 135–147. https://doi.org/10.1016/j.compfluid.2012.04.012

Andras Laszloffy, Jingping Long, and Abani K. Patra. 2000. Simple data management, scheduling and solution strategies for managing the irregularities in parallel adaptive *hp* finite element simulations. *Parallel Computing* 26, 13 (Dec. 2000), 1765–1788. https://doi.org/10.1016/S0167-8191(00)00054-5

Catherine Mavriplis. 1994. Adaptive mesh strategies for the spectral element method. *Computer Methods in Applied Mechanics and Engineering* 116, 1-4 (1994), 77–86. https://doi.org/10.1016/S0045-7825(94)80010-3

Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard (Version 4.0)*. Technical Report. University of Tennessee, Knoxville, TN. https://www.mpi-forum.org/.

William F. Mitchell. 1989. A comparison of adaptive refinement techniques for elliptic problems. *ACM Transactions on Mathematical Software* 15, 4 (Dec. 1989), 326–347. https://doi.org/10.1145/76909.76912

William F. Mitchell. 2002. The design of a parallel adaptive multi-level code in Fortran 90. In *Computational Science—ICCS 2002*. Lecture Notes in Computer Science, Vol. 2331. Springer, 672–680. https://doi.org/10.1007/3-540-47789-6_70

William F. Mitchell. 2007. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *Journal of Parallel and Distributed Computing* 67, 4 (April 2007), 417–429. https://doi.org/10.1016/j.jpdc.2006.11.003

William F. Mitchell. 2010. The *hp*-multigrid method applied to *hp*-adaptive refinement of triangular grids. *Numerical Linear Algebra with Applications* 17, 2-3 (April 2010), 211–228. https://doi.org/10.1002/nla.700

William F. Mitchell and Marjorie A. McClain. 2011. *A Comparison of hp-Adaptive Strategies for Elliptic Partial Differential Equations (Long Version)*. Technical Report 7824. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.IR.7824

William F. Mitchell and Marjorie A. McClain. 2014. A comparison of *hp*-adaptive strategies for elliptic partial differential equations. *ACM Transactions on Mathematical Software* 41, 1 (Oct. 2014), Article 2, 39 pages. https://doi.org/10.1145/2629459

Peter Munch, Timo Heister, Laura Prieto Saavedra, and Martin Kronbichler. 2022. Efficient distributed matrix-free multigrid methods on locally refined meshes for FEM computations. *arXiv:2203.12292 [cs, math]* (2022). https://doi.org/10.48550/arXiv.2203.12292

John Tinsley Oden, Abani Patra, and Yusheng Feng. 1994. Domain decomposition for adaptive *hp* finite element methods. In *Domain Decomposition Methods in Scientific and Engineering Computing*, David E. Keyes and Jinchao Xu (Eds.). Contemporary Mathematics, Vol. 180. American Mathematical Society, Providence, RI, 295–301. https://doi.org/10.1090/conm/180

Maciej Paszyński and Leszek Demkowicz. 2006. Parallel, fully automatic *hp*-adaptive 3D finite element package. *Engineering with Computers* 22, 3 (Dec. 2006), 255–276. https://doi.org/10.1007/s00366-006-0036-8

Maciej Paszyński and David Pardo. 2011. Parallel self-adaptive *hp* finite element method with shared data structure. *Computer Methods in Material Science* 11, 2 (2011), 399–405.

Abani Patra and John Tinsley Oden. 1995. Problem decomposition for adaptive *hp* finite element methods. *Computing Systems in Engineering* 6, 2 (April 1995), 97–109. https://doi.org/10.1016/0956-0521(95)00008-N

Will Pazner and Tzanio Kolev. 2022. Uniform subspace correction preconditioners for discontinuous Galerkin methods with *hp*-refinement. *Communications on Applied Mathematics and Computation* 4, 2 (June 2022), 697–727. https://doi.org/10.1007/s42967-021-00136-3

Yves Renard and Konstantinos Poulios. 2020. GetFEM: Automated FE modeling of multiphysics problems based on a generic weak form language. *ACM Transactions on Mathematical Software* 47, 1 (2020), Article 4, 31 pages.

Youcef Saad. 1993. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing* 14, 2 (March 1993), 461–469. https://doi.org/10.1137/0914028

David Silvester and Andrew Wathen. 1994. Fast iterative solution of stabilised Stokes systems part II: Using general block preconditioners. *SIAM Journal on Numerical Analysis* 31, 5 (Oct. 1994), 1352–1367. https://doi.org/10.1137/0731070

Pavel Šolín, Jakub Červený, and Ivo Doležel. 2008. Arbitrary-level hanging nodes and automatic adaptivity in the *hp*-FEM. *Mathematics and Computers in Simulation* 77, 1 (Feb. 2008), 117–132. https://doi.org/10.1016/j.matcom.2007.02.011

Shawn Strande, Haisong Cai, Mahidhar Tatineni, Wayne Pfeiffer, Christopher Irving, Amit Majumdar, Dmitry Mishin, et al. 2021. Expanse: Computing without boundaries: Architecture, deployment, and early operations experiences of a supercomputer designed for the rapid evolution in science and engineering. In *Practice and Experience in Advanced Research Computing*. ACM, New York, NY, 1–4. https://doi.org/10.1145/3437359.3465588

C. Taylor and P. Hood. 1973. A numerical solution of the Navier-Stokes equations using the finite element technique. *Computers & Fluids* 1, 1 (Jan. 1973), 73–100. https://doi.org/10.1016/0045-7930(73)90027-3

John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, et al. 2014. XSEDE: Accelerating scientific discovery. *Computing in Science & Engineering* 16, 5 (Sept. 2014), 62–74. https://doi.org/10.1109/MCSE.2014.80

Lin-Bo Zhang. 2019. *A Tutorial on PHG*. Technical Report. Academy of Mathematics and Systems Science, Chinese Academy of Sciences. http://lsec.cc.ac.cn/phg.

Received 10 June 2022; revised 27 April 2023; accepted 11 May 2023