



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

NON-CONFORMING MESH REFINEMENT FOR HIGH-ORDER FINITE ELEMENTS

J. Cervený, T. Kolev, V. Dobrev

May 24, 2018

SIAM Journal on Scientific Computing

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

NON-CONFORMING MESH REFINEMENT FOR HIGH-ORDER FINITE ELEMENTS*

JAKUB ČERVENÝ[†], VESELIN DOBREV[†], AND TZANIO KOLEV[†]

Abstract. We propose a general algorithm for non-conforming adaptive mesh refinement (AMR) of unstructured meshes in high-order finite element codes. Our focus is on h -refinement with a fixed p . The algorithm handles triangular, quadrilateral, hexahedral and prismatic meshes of arbitrarily high polynomial order, for any order finite element space in the de Rham sequence. We present a flexible data structure for meshes with hanging nodes and a general procedure to construct the AMR system through variational restriction. Our approach supports complex anisotropic refinement in both 2D and 3D, unlimited refinement levels of adjacent elements, derefinement, static condensation, hybridization, and MPI parallelism with load balancing based on space-filling curve partitioning. We report numerical experiments verifying the correctness of the algorithms, and perform a parallel scaling study to show that we can adapt meshes containing billions of elements and run efficiently on 393,000 parallel tasks. Finally, we illustrate the integration of dynamic AMR into a high-order Lagrangian hydrodynamics solver.

Key words. adaptive mesh refinement, non-conforming meshes, high-order finite elements, parallel computations, unstructured grids.

AMS subject classifications. 65N50, 65N30, 65Y05.

1. Introduction. High-order finite element methods (FEM) are becoming increasingly important in the field of scientific computing due to their increased accuracy and performance efficiency on modern computer architectures [11]. Many high-order applications are interested in parallel adaptive mesh refinement (AMR) on unstructured meshes containing quadrilateral and hexahedral elements. Such elements are attractive for their tensor product structure (enabling efficiency) and for their refinement flexibility (enabling *anisotropic* refinement). However, as opposed to bisection-based methods for local refinement of simplices, *hanging* nodes that occur after local refinement of quadrilaterals and hexahedra are not easily avoided by further refinement [32]. We are thus interested in handling *non-conforming* (irregular) meshes [14, 33, 19], in which adjacent elements need not share a complete face or edge and where some finite element degrees of freedom (DOFs) have to be constrained to obtain a conforming solution.

In this paper we present a set of software abstractions and algorithms for handling large parallel non-conforming meshes that we believe strikes the right balance between efficiency, ease of implementation and applicability to a wide range of problems. In particular, with a single general algorithm we handle a large class of 2D and 3D AMR meshes that consist of quadrilaterals and hexahedra, possibly mixed with triangles and triangular prisms. Generally, the elements can be refined by bisecting their edges, and we allow anisotropic refinement of tensor product elements, meaning that e.g. a hexahedron can be split into 2 or 4 hexahedra only. In addition, the method we propose is on a general discretization level, independent of the physics simulation, and supports the whole de Rham sequence of finite element spaces, at arbitrarily high-order. Our approach can also handle high-order curved meshes, as well as finite

*Performed under the auspices of the U.S. Department of Energy under Contract DE-AC52-07NA27344 (LLNL-JRNL-751849). The first author has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under the RICE New Technologies and Concepts for Smart Industrial Systems, project No. LO1607.

[†]Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory, Livermore, CA 94551, cerveny1@llnl.gov, dobrev1@llnl.gov, tzanio@llnl.gov

element techniques such as hybridization and static condensation [15]. While the algorithms we present can be extended to general hp -refinement, in this paper we focus exclusively on non-conforming h -refinement with a fixed order $p \geq 1$.

The methodology of constraining hanging nodes in non-conforming meshes is well established in the literature. Most authors opt to construct algebraic constraints that express constrained degrees of freedom as linear combinations of *true* degrees of freedom. The constraints are then applied to either the local element matrices as part of the global matrix assembly [1, 14, 9, 24, 5], after global assembly to “condense” the constrained DOFs out of the linear system [4], or, more rarely, at the shape function level [38] or the linear solver level [35]. Our approach is closest to the second method of adjusting the linear system, but we cast the elimination of constrained DOFs as a form of variational restriction. The practical advantage is that the stiffness matrix assembly stays unchanged and completely independent of the handling of AMR.

A large part of this paper is concerned with an efficient algorithm to construct a global operator that interpolates the constrained degrees of freedom from the true degrees of freedom. The algorithm is based on composing local interpolation matrices into a global one. It is easy to implement, works for a variety of finite element spaces and element geometries and is unique in being able to handle 3D anisotropic refinements as well as arbitrarily irregular meshes where adjacent elements are not limited to a 2:1 refinement ratio. The resulting interpolation matrix is also conveniently used to constrain duplicate nodes in parallel domain decomposition.

A modern finite element solver needs to scale to hundreds of thousands of parallel tasks. To achieve that, the mesh has to be fully distributed, as the global mesh no longer fits on each processor. Historically, this is not been the case in popular codes such as libMesh [24]. The deal.II project [4] has switched to the p4est library to distribute the mesh in 2011. In addition to mesh distribution, an AMR solver at this level of parallelism requires an extremely efficient partitioning and load balancing algorithm that can be applied often. High quality spectral type partitioners, such as ParMETIS [23], may be prohibitively expensive for this purpose. We thus take inspiration from octree-based approaches [17, 7, 21] where only the coarse elements (the octree roots) are shared by all parallel tasks and where the refinement tree provides a natural way to enumerate elements. The resulting sequence is equipartitioned and distributed among processors, which has been shown to be a scalable way to load balance the mesh [7]. In contrast to octree methods, however, we allow for more general refinement and support additional types of elements. Our partitioning algorithm is thus closest to the REFTREE method [30].

The rest of the paper is organized as follows: in Section 2, we define the class of meshes we support. In Section 3 we describe the general procedure to construct the AMR system through variational restriction. Section 4 is devoted to the algorithm for constructing the AMR interpolation operator, in serial. The internal data structures for element-based AMR in the context of high-order finite elements are presented in Section 5. Extending these methods to parallel settings is the subject of Section 6. Finally, Section 7 presents numerical experiments that illustrate the performance of our algorithms in practice.

2. Non-conforming meshes. For the purposes of this paper, and following the definitions in [29], an *element* K is a (closed) triangle, quadrilateral, triangular prism or hexahedron. An element contains *vertices*, *edges*, with the usual definitions, and three dimensional elements contain *faces*. A mesh $\{K_i\}_{i=1}^M$ is the union of M elements, not necessarily all of the same type, such that $\Omega = \cup_i K_i$ is a connected,

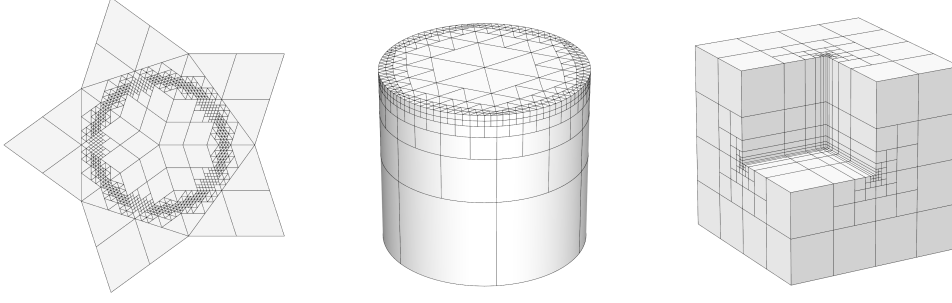


FIG. 2.1. Examples of non-conforming meshes handled by our algorithm: a mixed mesh containing triangles and quadrilaterals (left), a prismatic mesh obtained by refining 12 curved triangular prisms (middle), a hexahedral mesh after anisotropic refinement (right).

bounded region in \mathbb{R}^2 or \mathbb{R}^3 . Let $\langle K \rangle$ and ∂K denote the interior and boundary of element K . We assume that the element interiors do not intersect, i.e. $\langle K_i \rangle \cap \langle K_j \rangle = \emptyset$, for $i \neq j$. A mesh is said to be *conforming* if the set $\partial K_i \cap \partial K_j$, $i \neq j$, is a common vertex, common edge, common face or an empty set. A vertex of an element is called a *hanging node*, if it lies in the interior of an edge or face of another element. A mesh that contains at least one hanging node is called *non-conforming* or irregular.

To *refine* an element K_i means to replace it with at least two smaller elements K_{ik} of the same type, such that $K_i = \cup_k K_{ik}$. We call the elements K_{ik} *child elements* and K_i the *parent element*. The parent element is removed from the mesh after refinement, but it is stored along with links to the child elements in a *refinement tree*. In an inverse process called *derefinement*, the parent element can later be reintroduced to the mesh, once its child elements are removed.

When refining an element, we require that new vertices be placed in the center of edges or faces of the parent element. We thus only support bisection of edges during refinement. In this way a triangle can be split into 4 children, a quadrilateral into 2 or 4 children, and a prism or hexahedron into 2, 4 or 8 children. We refer to refinements that split all edges of the parent element as *isotropic* refinement, and the remaining refinement options as *anisotropic*. Some examples of the types of non-conforming meshes we support (obtained by refining initially conforming meshes) are shown in Figure 2.1.

A non-conforming mesh is said to be *consistent* if lower dimensional mesh entities (i.e., vertices, edges, faces) that intersect are either identical, or one is a proper subset of the other (see Figure 5.3 for an example of an inconsistent mesh). We refer to the smaller entity as a *slave* and the larger containing entity as a *master*. Lower dimensional entities of a consistent mesh that are neither masters nor slaves are called *conforming* entities.

Note that a slave face has exactly one master face, but a 3D edge can have multiple master edges. In that case we always take the smallest master edge, which in turn may be a slave to its smallest master. This hierarchy will end with a maximal master edge that is not a slave itself. Note that any master edge can be shared by several elements in a “conforming” manner. We call the interface between slave faces and their master, or slave edges and their maximal master the *coarse-fine interface*.

3. Non-conforming AMR by variational restriction. In the continuous Galerkin method [13], one starts with a weak variational problem: Find $u \in V$, such

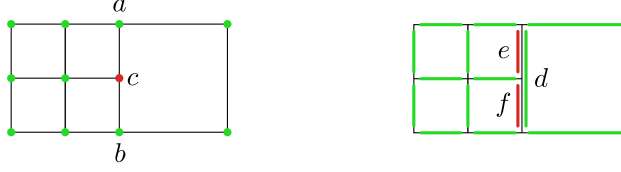


FIG. 3.1. Illustration of conformity constraints for lowest order elements in 2D. Left: nodal elements (H^1 subspace), constraint $c = (a + b)/2$. Right: Nedelec elements ($H(\text{curl})$ subspace), constraints $e = f = d/2$. In both cases, slave degrees of freedom are linearly interpolated from the degrees of freedom on the master side.

that

$$a(u, v) = l(v) \quad \forall v \in V, \quad (3.1)$$

where $a(\cdot, \cdot)$ and $f(\cdot)$ are bilinear and linear forms, respectively, on the inner-product space V , typically a Sobolev space on the domain Ω with imposed essential boundary conditions. The problem (3.1) is discretized by constructing an approximate finite dimensional subspace $V_h \subset V$, $\dim(V_h) = N$, on the mesh $\{K_i\}_{i=1}^M$ and by assembling the stiffness matrix A and load vector f , so that

$$v_h^T A u_h = v_h^T f \quad \forall v_h \in \mathbb{R}^N. \quad (3.2)$$

This is equivalent to the linear system $A u_h = f$, which yields the approximate solution vector u_h .

When constructing the space V_h , care must be taken to ensure that conformity requirements of the space V are met, so that V_h is a subspace of V . For example, if V is the Sobolev space H^1 , the functions in V_h must be kept continuous across element boundaries. If V is an $H(\text{curl})$ space, the tangential component of the finite element vector fields in V_h needs to be continuous across element faces, etc. In high-order FEM this is achieved by matching suitable *shape functions* at degrees of freedom associated with the mesh vertices, edges and faces, which together with DOFs in the element interiors form the N basis functions of V_h [37].

When the mesh is non-conforming, the shape functions cannot be matched directly at the coarse-fine interfaces. If we assign degrees of freedom in the usual way on the conforming entities (including vertices, and maximal master edges in 3D) and disregard any conformity in the interior coarse-fine interfaces, we obtain a larger space $\hat{V}_h \supset V_h$ that is “broken” along these interfaces, i.e., slave entities have DOFs that are independent of their master DOFs. We call \hat{V}_h the *partially conforming space*, with $\dim(\hat{V}_h) = \hat{N} > N$.

To restore conformity at the coarse-fine interfaces, the degrees of freedom on slave entities (slave DOFs) must be constrained so that the slave entities interpolate the finite element functions of their masters. Since the masters and slaves have matching function spaces (all elements have the same order p), the slave DOFs can simply be expressed as linear combinations of the master DOFs. Examples of such local constraints for low-order elements are shown in Figure 3.1.

Note that in general non-conforming meshes, some DOFs on the boundaries of master entities may themselves be constrained by another master entity, or a chain of them. In this paper we are concerned with *solvable* non-conforming meshes, where each such dependency chain ends with unconstrained degrees of freedom. The final

set of unconstrained DOFs is called *true* DOFs, denoted u_h . A solution vector \hat{u}_h in the partially conforming space \hat{V}_h can thus be written as

$$\hat{u}_h = \begin{pmatrix} u_h \\ w_h \end{pmatrix} \quad u_h \in \mathbb{R}^N, w_h \in \mathbb{R}^{(\hat{N}-N)}, \quad (3.3)$$

where w_h represents all slave DOFs and can be evaluated at any time by a linear interpolation $w_h = W u_h$ for some interpolation matrix W . We can also write

$$\hat{u}_h = P u_h, \quad \text{where } P = \begin{pmatrix} I \\ W \end{pmatrix}. \quad (3.4)$$

P will be called the *conforming prolongation matrix*. The N columns of P define the conforming basis functions of V_h , expressed as vectors in \hat{V}_h . We also use P to variationally restrict the solution \hat{u}_h to the conforming subspace V_h .

Since we want to upgrade an existing finite element code to support non-conforming meshes with as few modifications as possible, we let the code assemble the stiffness matrix \hat{A} and load vector \hat{f} in the space \hat{V}_h , as if there were no hanging nodes in the mesh. Of course, when computed in \hat{V}_h , the discrete problem (3.2) and the corresponding linear system $\hat{A} \hat{u}_h = \hat{f}$ result in a non-conforming solution where the slave DOFs are not constrained. However, taking $\hat{u}_h = P u_h$ and $\hat{v}_h = P v_h$, the variational formulation on V_h becomes

$$v_h^T P^T \hat{A} P u_h = v_h^T P^T \hat{f} \quad \forall v_h \in \mathbb{R}^N, \quad (3.5)$$

so we can solve the smaller system

$$P^T \hat{A} P u_h = P^T \hat{f}, \quad (3.6)$$

where the slave DOFs are eliminated. We then prolongate u_h to obtain a conforming solution $\hat{u}_h \in \hat{V}_h$.

An important point regarding essential boundary conditions in V_h is that they have to be applied in the system (3.6) *after* the variational transformation, i.e., based on the matrix $P^T \hat{A} P$ and right hand side $P^T \hat{f}$. This is in contrast to many non-AMR finite element settings, where one can eliminate the essential boundary conditions on element level. Indeed, it is straightforward to verify that for a general sub-matrix W , the elimination based on \hat{A} and \hat{f} followed by variational restriction is different than performing variational restriction and then elimination. In our experience, ensuring that the essential boundary conditions are eliminated at the very end is the main change needed in applications adopting the proposed AMR approach.

Another algebraic transformation that is commonly used in practice is static condensation, see e.g. [39]. Since static condensation manipulates DOFs in the interior of the element, and such DOFs are always true, AMR and static condensation commute, i.e., static condensation followed by AMR variational restriction is the same as AMR reduction followed by static condensation. The variational restriction approach can also be used to couple AMR with hybridization, see [15].

4. Constructing serial P . We now describe the algorithm to calculate the conforming prolongation matrix in a single-processor setting. As indicated in the previous section, factoring P out of the coupled AMR matrix makes it easier to incorporate AMR in applications. The explicit availability of P can also facilitate the development of AMR-specific linear solvers and preconditioners. Many codes apply

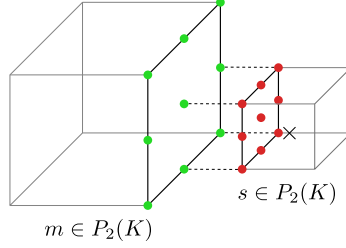


FIG. 4.1. Example of local constraining relations for quadratic hexahedral elements. The slave degrees of freedom, s , interpolate the master degrees of freedom, m . We can write $s = Qm$, where Q is a local interpolation matrix (in this case $Q \in \mathbb{R}^{9 \times 9}$).

P without assembling it, but in our case it is more convenient to build it explicitly due to the complex dependencies that can arise in arbitrarily irregular 3D meshes.

The P matrix is constructed for a concrete instance of a partially conforming finite element space \hat{V}_h . The procedure described below constructs both the true DOF identity I and the slave interpolation W submatrices of P . The first step is to determine the master-slave relations of non-conforming edges and faces. Recursive algorithms for that are described later, in Section 5.6. Here we will assume that we are given lists of master edges and faces and their associated slaves that are produced by these algorithms.

For each slave entity s_k , the position of its vertices within the reference domain of its master edge/face is known. For example, the vertices of the slave face in Figure 4.1 could have coordinates $\{(0.5, 0), (1, 0), (1, 0.5), (0.5, 0.5)\}$ in the reference domain of the master face. From these we can calculate the local interpolation matrix Q^k which expresses the slave DOFs (including DOFs on the boundary of the slave face) as linear combinations of the DOFs of the master face (including its boundary DOFs). In the case of nodal elements, the rows of Q^k contain just the shape functions of the master face finite element evaluated in the nodal positions of the slave face, i.e. the slave face performs nodal interpolation of the master face finite element function. Since the polynomial degrees of the two faces match, the interpolation is exact. Although we primarily use nodal elements, the local interpolation matrix is not limited to nodal elements and can be computed for any kind of finite element.

Note that sometimes a row of Q^k will be of the trivial form $Q_i^k = (0, \dots, 0, 1, 0, \dots, 0)$, which means that the slave DOF is identical to the master DOF and may not be constrained at all. This is the case, for example, for the node marked with “ \times ” in Figure 4.1. Since the matrices Q^k include the boundary DOFs of the slave face, certain dependencies will be computed multiple times — this is done to simplify the implementation at a relatively small computational cost.

As we iterate through the local interpolation matrices Q^k of all slave edges/faces, we continuously update a global dependency matrix D_{ij} , $1 \leq i, j \leq \hat{N}$. The matrix D , corresponding to the DOFs in \hat{V}_h , is represented as a sparse matrix and is initialized to identity. For each Q^k , we assign the local interpolation weights to D_{ij} for each slave DOF i that depends on a master DOF j . This allows us to identify three kinds of rows in the dependency matrix at the end of this phase:

1. *Identity rows*: $D_i = (0, \dots, 0, 1, 0, \dots, 0)$. These represent unconstrained, true degrees of freedom.

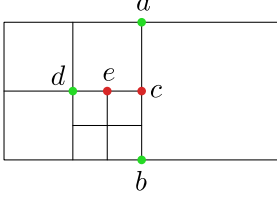


FIG. 4.2. *Example of indirect constraints: $e = (c + d)/2$ depends on $c = (a + b)/2$.*

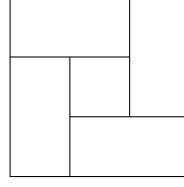


FIG. 4.3. *Unsolvable mesh with cyclic dependencies between nodal degrees of freedom.*

2. *Direct slave rows:* non-identity $D_i = (d_{ij})$ with nonzero entries d_{ij} corresponding only to unconstrained degrees of freedom j . These are slave degrees of freedom directly dependent on true DOFs.
3. *Indirect slave rows:* the remaining non-identity rows represent slave degrees of freedom dependent on true and/or other slave DOFs. These rows appear because we do not restrict the mesh regularity, see Figure 4.2 and the discussion below.

Let N be the number of identity rows of the D matrix. This is the number of true degrees of freedom and the dimension of the conforming finite element space V_h . We initialize P as an $\hat{N} \times N$ sparse matrix with identity in the N rows corresponding to the true DOFs. To fill in the remaining rows, we introduce a Boolean vector r of size \hat{N} , which keeps track of which rows in P have been *resolved*. If i is a true DOF, we initialize $r_i = 1$, otherwise we set $r_i = 0$. We loop over the rows of D , scanning for rows where $r_i = 0$ and $r_j \neq 0$ for all nonzero D_{ij} . These rows correspond to unresolved constrained DOFs that depend on already resolved DOFs. When such D_i is found, the (sparse) row P_i is set to

$$P_i = \sum_j D_{ij} P_j, \quad (4.1)$$

and the resolution is marked by setting r_i to one.

This procedure is iterated until all unresolved P_i are assigned a linear combination of already known P_j . In the first sweep, all direct dependencies are resolved (e.g. the dependence of c on a and b in Figure 4.2), which unlocks indirectly dependent slave DOFs that are resolved in subsequent sweeps (e.g. the dependence of e on c in Figure 4.2). The number of sweeps needed is based on the longest indirect dependency in the non-conforming mesh known as *irregularity* [2, 14, 10]: 1-irregular meshes have only one level of dependency, while n -irregular meshes will require n sweeps to resolve all rows of P . In practice, we typically have at most 3-irregular meshes, so only a few sweeps are needed for all rows of P to be resolved.

The algorithm is only guaranteed to finish if there are no cyclic dependencies, i.e., if the mesh is solvable. In the serial algorithm we include an explicit check: if there are still unresolved rows i but none of them has $r_j \neq 0$ for all nonzero D_{ij} , we print an error message. This never happens if the AMR process starts on an initially conforming (regular) mesh. However, it is possible to construct a non-conforming mesh with cyclic dependencies. Figure 4.3 shows a mesh (inspired by [10]) that is representable by our data structure but leads to an infinite loop in the unchecked P matrix algorithm. Note that in this case, a single level of additional refinement breaks the cycle and makes the mesh solvable.

5. Serial data structures. In this section we describe how we represent non-conforming meshes and how we determine master-slave relations needed for the construction of P . A high-order FEM code needs to be able to track vertices, edges and faces and their relations to incident elements. Octree-based algorithms can derive this connectivity from the structure of the tree itself, though on octree boundaries the algorithms get complex (see interoctree connectivity in [7]). Since our meshes are more general than octrees, we take a more traditional approach with explicit links between mesh entities. A comprehensive and mathematically elegant approach is the Sieve representation [25], storing a graph with edges connecting entities whose dimension d differs by one. However, this cell-complex representation does not generalize naturally to non-conforming meshes [22] and stores more information than we need.

As an alternative, we merely connect each $d > 0$ dimensional entity to its vertices and store the inverse mappings (from vertices back to edges and faces) as hash tables. This simple data structure provides the mesh connectivity we need and allows us to traverse the non-conforming interfaces. More specifically,

- edges and hanging vertices are identified by pairs of vertex indices,
- faces are identified by four vertex indices, and contain two element references,
- elements contain indices of up to 8 vertices, or indices of children if refined.

For a uniform hexahedral mesh, this representation requires about 290 bytes per element, counting the vertices, edges, faces and their 32-bit indices, and including the hash tables and the refinement trees. For a high-order code we believe this to be acceptable, as a high-order mesh typically contains at least 8 times fewer elements than a comparable low-order mesh.

5.1. Refinement trees. Since we aim for dynamic AMR capable of derefinement, we need to store the refinement history. When the elements of the initial mesh get refined, they are removed from the list of active elements and serve as roots of refinement trees. A refined element points to its children instead of its vertices, as shown in pseudocode in Figure 5.1. To save memory, we use 32-bit integers to identify entities instead of pointers. An element can have fewer than 8 children if it is refined anisotropically. We store a computed list of all leaf nodes of all refinement trees. This list represents the current state of the mesh.

5.2. Hash maps. The mapping from vertices to other entities is done by hash tables. We illustrate this on an example shown in Figure 5.2. Assume (v_1, v_2, v_3, v_4) is a 3D face. Across this face there are three neighboring elements with faces (v_1, v_5, v_8, v_7) , (v_5, v_2, v_3, v_6) and (v_7, v_8, v_6, v_4) . To access edges and faces from the incident elements, we can call

```

 $e_1 \leftarrow \text{GetEdge}(v_1, v_5)$ 
 $e_2 \leftarrow \text{GetEdge}(v_5, v_2)$ 
 $f_1 \leftarrow \text{GetFace}(v_1, v_5, v_8, v_7)$ 
 $f_2 \leftarrow \text{GetFace}(v_5, v_2, v_3, v_6)$ 
...
```

A similar hash map exists for hanging vertices and is used to traverse non-conforming interfaces in Section 5.6:

```

 $v_5 \leftarrow \text{GetVertex}(v_1, v_2)$ 
 $v_6 \leftarrow \text{GetVertex}(v_4, v_3)$ 
 $v_7 \leftarrow \text{GetVertex}(v_1, v_4)$ 
 $v_8 \leftarrow \text{GetVertex}(v_5, v_6)$ 
```

Each Get function searches the respective hash table and returns a 32-bit entity in-

```

struct ELEMENT
  bool refined
  union
    int vertex[8]
    int child[8]
  end union
  int parent
  int rank      ▷ see Section 6
end struct

```

FIG. 5.1. Data structure for both refined and leaf elements (pseudocode).

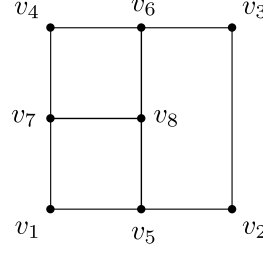


FIG. 5.2. Example of a hexahedron face (v_1, v_2, v_3, v_4), adjacent to three neighboring hexahedra in a non-conforming mesh.

dex. The order of arguments is not relevant, i.e., $\text{GetVertex}(v_1, v_2) = \text{GetVertex}(v_2, v_1)$. We use the convention that the Get functions create the requested entity if it does not exist. To query its existence, each GetX function has a FindX counterpart taking the same arguments and returning **nil** (encoded as -1) if the entity does not exist, instead of creating it. We make sure the hash tables are properly sized to get amortized $O(1)$ complexity of each query.

5.3. Refinement. To refine e.g. a hexahedron, we first create new mid-edge vertices (or get already existing ones) by calling $\text{GetVertex}(\cdot, \cdot)$ for all 12 edges. Six mid-face vertices are accessed through opposite mid-edge vertices; the two options per face must be checked first with $\text{FindVertex}(\cdot, \cdot)$ to prevent creating a duplicate mid-face vertex, in case it already exists. The mid-element vertex is accessed from any two opposite mid-face vertices. New elements are created with indices of the new vertices. The old element is marked as refined and its vertex indices are replaced with indices of the child elements.

Reference counting is used to keep track of the number of elements pointing to each vertex, edge and face. If the reference count drops to zero, the entity is deleted.

5.4. Anisotropic refinement. Our data structure allows for anisotropic refinement. The challenge is maintaining consistency of the mesh when refining anisotropically. It is easy to refine adjacent elements in such a way that their faces are not subsets or supersets of each other, so there is no way to determine a master-slave relationship. This is illustrated in Figure 5.3. To resolve such a situation, the neighbor of the element being refined has to be refined as well. Such *forced* refinements may propagate, until a globally consistent mesh is obtained. We defer the treatment of this ripple effect to a follow-up publication.

5.5. Derefinement. When derefining an element, the indices of its original vertices are first retrieved from the children (since the refinement pattern is known). All of the children need to be leaves, otherwise they are recursively derefined first. In an inverse process to refinement, the child elements are destroyed and associated vertex, edge and face reference counts are decremented. The coarse element is reactivated, and the appropriate reference counts are incremented.

5.6. Calculating master-slave relations. We loop over all leaf elements and their faces. Each face, triangular or quadrilateral, is checked if it is a master face by invoking the function TraverseTriFace (Algorithm 1) or TraverseQuadFace (Algorithm 2). These functions attempt to recursively descend an implicit face refinement “tree” and return a list of slave faces, if they can be reached, or an empty list if the starting

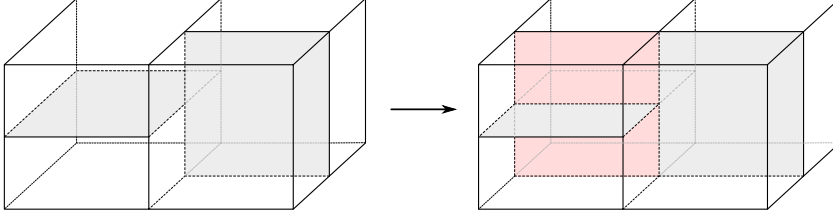


FIG. 5.3. *Anisotropic refinements may lead to an inconsistent mesh (left). Additional refinements are needed to restore valid master-slave relations between faces (right).*

face is conforming. The function `FindVertex` is used here to obtain vertices that were created at the center of edges of previously refined elements. During descent, the position within the master face’s reference domain is tracked by the points p_i , which are initialized to the reference domain corners (at $depth = 0$).

In the case of quadrilateral faces, the face refinement “tree” is traversed as a binary tree, i.e., isotropically refined faces are treated as if refined anisotropically twice (horizontally and then vertically relative to their reference domain, or vice versa). The function `FaceSplitType` (Algorithm 3) determines which way a face is split, again using `FindVertex` to look up mid-edge and mid-face vertices (refer to Figure 5.2 for a quadrilateral face example).

Algorithm 1 List slave faces of a potential triangular master face.

```

1: function TRAVERSETRIFACE( $v_1, v_2, v_3, p_1, p_2, p_3, depth$ )
2:    $v_{12} \leftarrow \text{FindVertex}(v_1, v_2), p_{12} \leftarrow (p_1 + p_2)/2$ 
3:    $v_{23} \leftarrow \text{FindVertex}(v_2, v_3), p_{23} \leftarrow (p_2 + p_3)/2$ 
4:    $v_{31} \leftarrow \text{FindVertex}(v_3, v_1), p_{31} \leftarrow (p_3 + p_1)/2$ 
5:    $list \leftarrow []$ 
6:   if ( $v_{12}$  not nil) and ( $v_{23}$  not nil) and ( $v_{31}$  not nil) then
7:      $list \leftarrow \text{TraverseTriFace}(v_1, v_{12}, v_{31}, p_1, p_{12}, p_{31}, depth + 1) +$ 
8:        $\text{TraverseTriFace}(v_{12}, v_2, v_{23}, p_{12}, p_2, p_{23}, depth + 1) +$ 
9:        $\text{TraverseTriFace}(v_{31}, v_{23}, v_3, p_{31}, p_{23}, p_3, depth + 1) +$ 
10:       $\text{TraverseTriFace}(v_{12}, v_{23}, v_{31}, p_{12}, p_{23}, p_{31}, depth + 1)$ 
11:   else if  $depth > 0$  then
12:      $slave \leftarrow \text{FindFace}(v_1, v_2, v_3, \text{nil})$ 
13:     if  $slave$  not nil then
14:        $list \leftarrow [\{slave, p_1, p_2, p_3\}]$ 
15:     end if
16:   end if
17:   return  $list$ 
18: end function

```

An analogous function, `TraverseEdge`, is employed to construct the list of master-slave relations for edges. Edges of each leaf element are checked for slave edges by traversing the local vertex hierarchy in a similar recursive manner.

The 2D case is treated with the same code as a degenerate case: triangular and quadrilateral elements merely have a different number of vertices/edges in internal geometry tables and they have zero faces, so creation and traversal of faces is skipped.

We make no assumptions regarding the level of refinement of adjacent elements.

Algorithm 2 List slave faces of a potential quadrilateral master face.

```

1: function TRAVERSEQUADFACE( $v_1, v_2, v_3, v_4, p_1, p_2, p_3, p_4, depth$ )
2:    $split \leftarrow \text{FaceSplitType}(v_1, v_2, v_3, v_4)$ 
3:    $list \leftarrow []$ 
4:   if  $split = \text{Vertical}$  then
5:      $v_{12} \leftarrow \text{FindVertex}(v_1, v_2), p_{12} \leftarrow (p_1 + p_2)/2$ 
6:      $v_{34} \leftarrow \text{FindVertex}(v_3, v_4), p_{34} \leftarrow (p_3 + p_4)/2$ 
7:      $list \leftarrow \text{TraverseQuadFace}(v_1, v_{12}, v_{34}, v_4, p_1, p_{12}, p_{34}, p_4, depth + 1) +$ 
8:        $\text{TraverseQuadFace}(v_{12}, v_2, v_3, v_{34}, p_{12}, p_2, p_3, p_{34}, depth + 1)$ 
9:   else if  $split = \text{Horizontal}$  then
10:     $v_{23} \leftarrow \text{FindVertex}(v_2, v_3), p_{23} \leftarrow (p_2 + p_3)/2$ 
11:     $v_{41} \leftarrow \text{FindVertex}(v_4, v_1), p_{41} \leftarrow (p_4 + p_1)/2$ 
12:     $list \leftarrow \text{TraverseQuadFace}(v_1, v_2, v_{23}, v_{41}, p_1, p_2, p_{23}, p_{41}, depth + 1) +$ 
13:       $\text{TraverseQuadFace}(v_{41}, v_{23}, v_3, v_4, p_{41}, p_{23}, p_3, p_4, depth + 1)$ 
14:   else if  $depth > 0$  then
15:      $slave \leftarrow \text{FindFace}(v_1, v_2, v_3, v_4)$ 
16:      $list \leftarrow [\{slave, p_1, p_2, p_3, p_4\}]$ 
17:   end if
18:   return  $list$ 
19: end function

```

Algorithm 3 Determine whether a face is split horizontally, vertically or not split.

```

1: function FACESPLITTYPE( $v_1, v_2, v_3, v_4$ )
2:    $v_{12} \leftarrow \text{FindVertex}(v_1, v_2)$ 
3:    $v_{23} \leftarrow \text{FindVertex}(v_2, v_3)$ 
4:    $v_{34} \leftarrow \text{FindVertex}(v_3, v_4)$ 
5:    $v_{41} \leftarrow \text{FindVertex}(v_4, v_1)$ 
6:    $midf_1 \leftarrow (v_{12} \text{ not nil and } v_{34} \text{ not nil}) ? \text{FindVertex}(v_{12}, v_{34}) : \text{nil}$ 
7:    $midf_2 \leftarrow (v_{23} \text{ not nil and } v_{41} \text{ not nil}) ? \text{FindVertex}(v_{23}, v_{41}) : \text{nil}$ 
8:   if  $midf_1$  is nil and  $midf_2$  is nil then
9:     return NotSplit
10:  else
11:    return  $(midf_1 \text{ not nil}) ? \text{Vertical} : \text{Horizontal}$ 
12:  end if
13: end function

```

The simple algorithms above work for general non-conforming 3D meshes, including meshes with anisotropic refinements.

5.7. Element neighbors. The parallel algorithms in Section 6 require the ability to determine element neighbors. We define two elements to be neighbors if their closed sets have a non-empty intersection, even if the intersection is just a single vertex (i.e. we consider all of the vertex-, edge- and face-neighbors). Although algorithms exist to determine such neighbors from the structure of the refinement tree itself, the handling of multiple trees may become complex [7].

Instead, we employ an approach based on a Boolean matrix of element-vertex connectivity. Let B be a Boolean matrix containing one row per element and one column per mesh vertex. For each element, its row marks vertices that belong to the element, including possible mid-edge and mid-face vertices of neighboring refined

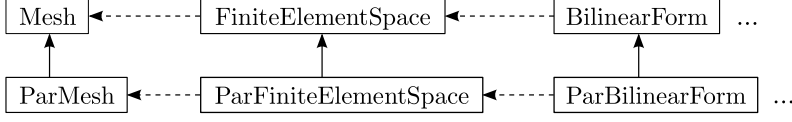


FIG. 6.1. *Parallel classes inherit from, and partially override, serial classes.*

elements. The incident vertices are collected for each element by a procedure similar to Algorithms 1 and 2. Then, if e is a Boolean vector representing the set of elements whose neighbors we need to find, the vector $e' = BB^T e$ will represent the original set of elements plus all its neighbors. The BB^T product is not stored, only its action is computed on demand. To save even more memory, the obvious corner vertices of each element are also not explicitly stored in B . The matrix is thus empty if the mesh is conforming.

6. Parallelization. In this section we discuss parallelization of the serial AMR algorithms and data structures described in Sections 4 and 5. Motivated by the implementation in the MFEM finite element library [28], we propose an approach that handles MPI parallelism as an additional layer that reuses as much of the serial code as possible. In terms of object-oriented design, this is done by sub-classing the serial classes to augment them with parallel logic (see Figure 6.1), occasionally overriding small parts of the code using virtual functions.

We split the mesh into K disjoint regions (see next section), where K is the number of MPI tasks. The partitioning is element based, i.e., each element is assigned to one of the tasks k . The vertices, edges and faces on the boundary of each task's region are duplicated, so that each region can be treated as an isolated mesh and passed to the serial part of the code. Repeating the approach from Section 3, we assemble the stiffness matrices \hat{A}_k and load vectors \hat{f}_k locally on each MPI task k , as if the mesh was conforming and serial on each task.

Globally, we have a new, parallel finite element space $\tilde{V}_h \supset \hat{V}_h$, with $\tilde{N} = \dim \tilde{V}_h > \dim \hat{V}_h = \tilde{N}$. The parallel stiffness matrix \tilde{A} consists of diagonal blocks \hat{A}_k and the parallel load vector \tilde{f} contains the blocks \hat{f}_k . Again, the solution of the parallel system $\tilde{A}\tilde{u}_h = \tilde{f}$ would be disconnected along the interfaces of the K regions, but we can use the same variational restriction approach to obtain a globally conforming parallel solution. The parallel conforming prolongation matrix, still denoted P , is now of the size $\tilde{N} \times N$ and conveniently handles both conforming interpolation and parallel decomposition, by mapping from the space V_h directly to \tilde{V}_h .

In the current version of MFEM, we explicitly form the parallel triple matrix product $P^T \tilde{A} P$ using the *RAP* triple-matrix-product kernel from the *hypre* library [20]. This kernel is highly optimized in *hypre* as it is used internally for the coarse-grid operator construction in its algebraic multigrid solvers. In its next release, MFEM will also support the direct evaluation of the action of $P^T \tilde{A} P$, without assembly, which is more efficient for higher order elements.

6.1. Tree-based partitioning. Similarly to [18, 8, 36, 30, 7, 6], we partition the mesh by splitting the space-filling curve (SFC) obtained by enumerating depth-first all leaf elements of all refinement trees. We assume that the elements of the coarse mesh are ordered as a sequence of face-neighbors, so that a globally continuous SFC can be obtained. For ordering the coarse mesh (i.e., the refinement tree roots) we use the Gecko library [27]. Once the coarse mesh is ordered we invoke depth-first

state	child order	child state
0	(0, 1, 2, 3)	(1, 0, 0, 5)
1	(0, 3, 2, 1)	(0, 1, 1, 4)
2	(1, 2, 3, 0)	(3, 2, 2, 7)
3	(1, 0, 3, 2)	(2, 3, 3, 6)
4	(2, 3, 0, 1)	(5, 4, 4, 1)
5	(2, 1, 0, 3)	(4, 5, 5, 0)
6	(3, 0, 1, 2)	(7, 6, 6, 3)
7	(3, 2, 1, 0)	(6, 7, 7, 2)

FIG. 6.2. Ordering of subtrees to obtain the Hilbert curve in 2D.

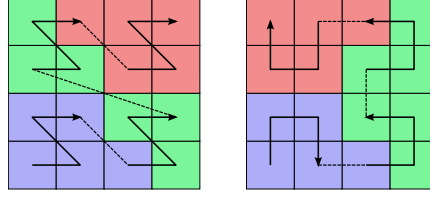


FIG. 6.3. Example of Z-curve (left) vs. Hilbert curve partitioning.

traversal for the ordered and oriented roots. Orienting a root element means assigning an initial traversal state (see below) so that SFCs of successive trees are connected.

In the case of quadrilaterals and hexahedra, the simplest traversal with a fixed order of children at each tree node leads to the well-known Z-curve, as shown in Figure 6.3 (left). However, it turns out that simply by changing the order of visiting subtrees at each level, the Hilbert curve can be obtained at no additional cost. The Hilbert curve produces continuous partitions and leads to better interprocess connectivity [8] than the Z-curve. In the simpler 2D case, each tree node can be in 8 states (4 rotations times 2 inversions). Given a state of the root node, we descend to subtrees in the appropriate order, passing a new state to each, according to Table 6.2. Example of a Hilbert curve partitioning is shown in Figure 6.3 (right). The method is analogous for hexahedra, where a tree node has 24 states. The corresponding ordering table can be found in [8] or in our implementation [28].

Generating a continuous SFC is more difficult for triangles, prisms and in the presence of anisotropic refinements. The Sierpinski curve is optimal for triangles [30], but we would have to switch to triangle bisection. Prisms admit a continuous SFC analogous to hexahedra, but not all elements can be face neighbors (only edge neighbors). The most serious difficulty is with anisotropic refinement, which probably prevents a continuous SFC, and generating a good leaf sequence is an open problem.

At the beginning of the computation, we assume that the mesh is small enough to fit completely in the memory of each MPI task. Each task uses a copy of the serial mesh and traverses its trees as described. The list of leaf elements is split into equal sized parts and each part is assigned to one MPI task by setting the members `Element::rank`. Each task thus owns a region of the mesh formed by elements of the corresponding rank. The region may not have a minimal surface, but it should be continuous and relatively compact due to the SFC locality.

6.2. Ghost layer. Although the serial code sees a non-overlapping decomposition of the mesh, in the spirit of [7], we internally hold the (pruned) refinement trees on all MPI tasks. For the purpose of constructing P , we also track *ghost elements* for each MPI task. A ghost element is one that is a vertex-, edge-, or face-neighbor to an element owned by an MPI task, but is itself owned by another task. The set of all ghost elements forms a minimal layer of elements enclosing a task's region. This layer needs to be kept synchronized with neighboring tasks.

On each task, elements beyond the ghost layer may not correspond to real elements, and we prune the refinements in this area immediately after partitioning the

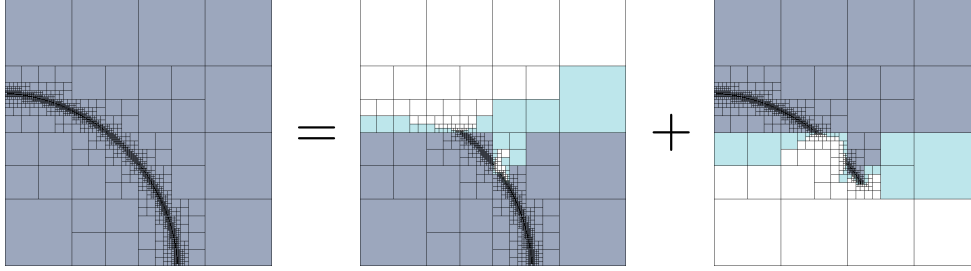


FIG. 6.4. A 2D mesh partitioned by the Hilbert curve between two MPI tasks. Light-gray elements are ghost elements. White areas have been pruned and do not represent real elements.

serial mesh and identifying the ghost elements. Pruning the refinement tree means removing (derefining) all subtrees that contain only leaves not owned by the current task nor belonging to the ghost layer [3]. After this step, the mesh becomes fully distributed as no single task sees all of the actual leaf elements. This is illustrated in Figure 6.4 for two tasks in 2D.

The last type of elements we identify are parallel boundary elements. This layer is a subset of the region owned by each MPI task formed by elements which are vertex-, edge-, or face-neighbors to the ghost layer. The boundary elements constitute ghost elements of one or more neighboring tasks and any change in them needs to be communicated to the neighbors to keep all the ghost layers synchronized. For example, if a refinement occurs in one task's boundary layer, a message needs to be sent to all neighboring tasks that share the elements, so they can duplicate the refinement in their ghost layer.

6.3. Load balancing. Balancing the mesh so that each MPI task has the same number of elements (± 1 if the total number of elements is not divisible by the number of tasks) is relatively straightforward in the context of SFC based partitioning. A parallel partial sum (`MPI_Scan`) is done to determine new beginnings of the partitions in the global sequence of leaf elements. New assignments of `Element::rank` within each parallel region can then be computed. Next, the new assignments within the ghost layers must be communicated between neighboring tasks (this is the most expensive part). This facilitates the final step, in which elements no longer assigned to the current task are sent to the new owners, together with a layer of ghost elements, to ensure a valid new state of the distributed mesh. The exchange of elements ends when each task obtains the right number of elements, which is known beforehand. After the load balancing procedure, a pruning step is done on each task to remove branches of the refinement tree that no longer need to be represented, i.e., subtrees that only contain leaves beyond the ghost layer are removed.

6.4. Message encoding. The prime advantage of storing the coarse mesh (i.e., all refinement tree roots) on all MPI tasks is the ability to refer to any element of the global mesh by the index of a root element and the refinement path to the element in question. However, rather than identify elements in this way individually, we develop an algorithm to encode a *subtree* of the global refinement hierarchy, since our messages usually carry information for multiple elements at once.

Given a set of leaf elements in a local mesh, we use the member `Element::parent` recursively to identify the set of trees participating in the element set and store their indices. In each tree, we then descend along the same paths back to the leaves. At

each node, we output an 8-bit mask indicating which subtrees contain elements from the set. We output a zero mask to terminate the descent at leaves.

The functionality of encoding element sets is used for load balancing, but also as a basis for encoding sets of vertices, edges, faces and their degrees of freedom, as required in Section 6.5. A vertex, edge or face can be identified by the index of an element in an element set, followed by the local number of the vertex/edge/face within that element. A DOF is identified by its index within its mesh entity.

We use variable length `MPI_BYTE` messages and try to collect as much data as possible for communication between each pair of tasks in order to minimize the total number of messages.

6.5. Constructing parallel P . The algorithm to build the P matrix in parallel is more complex, but conceptually similar to the serial algorithm. We still express slave DOF rows of P as linear combinations of other rows; however, some of them may be located on other MPI tasks and must be communicated first.

The ghost layer allows us to determine, without communication, which vertices, edges and faces are shared by more than one task. As we traverse the local mesh and its ghost layer using the serial algorithms of Section 5.6, we build a list of groups of MPI ranks, which we call communication groups. Each vertex, edge and face is assigned a communication group index, and an owner rank. In a group, the task with the lowest rank is defined to be the owner. We store the owner explicitly for each mesh object, because as the next step, we modify the communication groups so that master and slave faces and edges are grouped together. This is to ensure that information will flow from the owners and masters to non-owners and slaves.

Next, for a concrete finite element space, we assign \hat{N}_k local degrees of freedom on each MPI task k . These are the DOFs of all elements of the task's region, whether slave, master or conforming. Globally, $\sum \hat{N}_k = \hat{N}$. Each local DOF has a unique owner, inherited from its underlying vertex, edge or face. DOFs with a remote owner are expected to later receive a row of the P matrix that will make them identical to a remote true DOF.

In addition to local DOFs, we also assign \hat{G}_k virtual DOFs within the ghost layer (if not already marked as local), all with indices greater than \hat{N}_k . We define vectors o, g, r of size $\hat{N}_k + \hat{G}_k$, where for each DOF i : o_i is the owner rank, g_i is the communication group index and r_i is the resolution flag, as in Section 4.

On each task we define a local dependency matrix D_{ij} , this time $1 \leq i \leq \hat{N}_k$ and $1 \leq j \leq \hat{N}_k + \hat{G}_k$, initialized to identity on the \hat{N}_k block. Reusing much of the serial code, we again collect the slave interpolation weights D_{ij} for each slave DOF i that depends on master DOFs j . Note that j may now be a ghost DOF, $j > \hat{N}_k$. Still without any communication, we can identify the following types of rows D_i :

1. *Identity rows*, with $o_i = k$. These are true DOFs that we own.
2. *Identity rows*, with $o_i \neq k$. Here i is a true DOF owned by another rank.
3. *Local slave rows*, non-identity D_i with $j \leq \hat{N}_k$ for all nonzero D_{ij} . These are slave DOFs dependent only on local DOFs.
4. *Remote slave rows*, non-identity D_i with at least one nonzero D_{ij} for $j > \hat{N}_k$. To resolve these slave DOFs, one or more remote P rows need to be received first.

Let N_k be the number of rows of type 1 on task k . We perform a parallel partial sum (`MPI_Scan`) on N_k to determine the global index $N_{0,k}$ of the first true DOF for

Algorithm 4 Construct parallel P matrix.

```

1: function CONSTRUCTPARALLELP(mesh, space)
2:   assign  $\hat{N}_k + \hat{G}_k$  local DOFs
3:   calculate local dependency matrix  $D_{ij}$  and vectors  $o_i, g_i$ 
4:    $N_k \leftarrow$  number of true DOFs owned
5:    $N_{0,k} \leftarrow$  MPI_Scan on  $N_k$   $\triangleright$  first column of our  $P$  partition
6:    $r_i \leftarrow 0$  for all  $i$ 
7:   for all  $N_k$  owned true DOFs  $i$  do
8:      $P_i \leftarrow$  identity, 1 at global column  $N_{0,k} + j$ ,  $0 \leq j < N_k$ 
9:      $r_i \leftarrow 1$ 
10:    add  $P_i$  to outgoing messages to ranks  $r \neq k$  of group  $g_i$ 
11:   end for
12:   repeat  $\triangleright$  main outer loop
13:     MPI_Isend all nonempty outgoing messages
14:     while MPI_Iprobe for some rank  $r$  do
15:       decode message from  $r$ , in particular the DOF numbers  $i$ 
16:       for all decoded DOFs  $i$  do
17:          $P_i \leftarrow$  row from message
18:          $r_i \leftarrow 1$ 
19:         add  $P_i$  to outgoing messages to other ranks if needed (see text)
20:       end for
21:     end while
22:     while exists  $i$  s.t.  $r_i = 0$  and  $r_j \neq 0$  for nonzero  $D_{ij}$  do  $\triangleright$  inner loop
23:        $P_i \leftarrow \sum_j D_{ij} P_j$ 
24:        $r_i \leftarrow 1$ 
25:       add  $P_i$  to outgoing messages to ranks  $r \neq k$  of group  $g_i$ 
26:     end while
27:   until  $r_i = 1$  for all  $i$ 
28:   return  $P$ 
29: end function

```

each rank, i.e. the column partitioning of P . We then run a version of the iterative algorithm of Section 4, wrapped in one more outer loop. The outer loop communicates remote rows of P when there is nothing more to be done by the *serial* (local) inner loop. In the first outer iteration, the inner loop resolves all DOFs that do not depend on remote DOFs. Then we send the rows of all DOFs that have just been resolved to neighboring tasks. The group g_i of a DOF determines which ranks need to be sent its P row. We collect the rows and send a single message per neighbor. The messages are decoded by the recipients and the received rows, containing global column numbers, are assigned to appropriate ghost DOFs. This unlocks (by setting the ghost DOF r_i) in the next outer iteration more DOFs that have been waiting for remote data. The complete procedure is summarized in Algorithm 4.

Situations may occur in which a ghost layer in an MPI task does not provide complete information on the ranks that depend on a particular master DOF. One such setup is shown in Figure 6.5, where DOF a is owned by rank 0, and the corresponding row P_a is needed by both rank 1 and rank 2 (e.g., to constrain DOF b owned by rank 2). The group g_a should therefore be $\{0, 1, 2\}$, but rank 0 does not see rank 2 elements—these are beyond its ghost layer. Since rank 0 sees the group $\{0, 1\}$, the

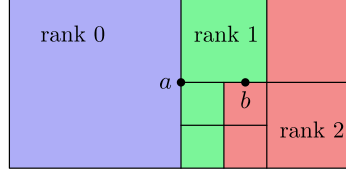


FIG. 6.5. Message from rank 0 to rank 1 needs to be forwarded to rank 2.

message is only sent to rank 1. However, rank 1 sees the correct group, and so it forwards the message to rank 2. This is accomplished by line 19 of Algorithm 4, which compares the sender's version of the group (encoded in the message) with the recipient's own view, and the message is sent to the missing ranks. To prevent infinite loops, a message is only allowed to be forwarded once. Testing shows that the number of rows that need forwarding is rarely more than 1%, typically in cases where the number of elements per task is small.

7. Numerical results. In this section we present numerical experiments that illustrate the performance of our unstructured AMR algorithms in practice. The results were obtained with the open-source implementation of the proposed methods in the MFEM finite element library [28].

7.1. Model problems. As a first test, we solve a standard AMR benchmark problem with a known exact solution, described in [31]. The goal is to reveal potential errors in the prolongation matrix algorithm by checking the behavior of the convergence curves. We also examine the benefit of AMR compared to uniform refinement.

The problem is the Poisson's equation $-\Delta u = f$ on the unit square $\Omega = (0, 1)^2$ with a Dirichlet boundary condition. The right hand side f is chosen so that the exact solution is

$$u(x, y) = \arctan \left(\alpha \left(\sqrt{(x - x_c)^2 + (y - y_c)^2} - r \right) \right). \quad (7.1)$$

The solution has a sharp circular wave front of radius r centered at (x_c, y_c) , as shown in Figure 7.1 (left). The parameters are $\alpha = 200$, $(x_c, y_c) = (-0.05, -0.05)$ and $r = 0.7$. We start with a mesh consisting of 4×4 squares and use nodal finite elements of degree $p = 1, 2, 4, 8$ (a subspace of H^1) to discretize the problem. We then perform an adaptation loop consisting of the following steps:

1. Solve the problem on the current mesh to get an approximate solution u_h .
2. For each element K_i , integrate the energy norm of the exact error,

$$e_i = ||u_h - u||_{E, K_i}. \quad (7.2)$$

3. Record the total error $e = (\sum_i e_i^2)^{1/2}$ and the current number of DOFs.
4. Refine elements for which $e_j > 0.7 \max_i \{e_i\}$.

The errors must be calculated with a sufficiently high order integration rule (we used a rule accurate for order 30), especially in the first few iterations. Figure 7.1 (center) shows the mesh after 11 iterations. The convergence curves are graphed for $p = 1, 2, 4, 8$ in Figure 7.2 (left; solid lines). We observe that the convergence histories exhibit the expected behavior.

For comparison with a previous (non-AMR) version of our code, we also plot convergence curves for the case of uniform refinement, meaning we refine *all* elements

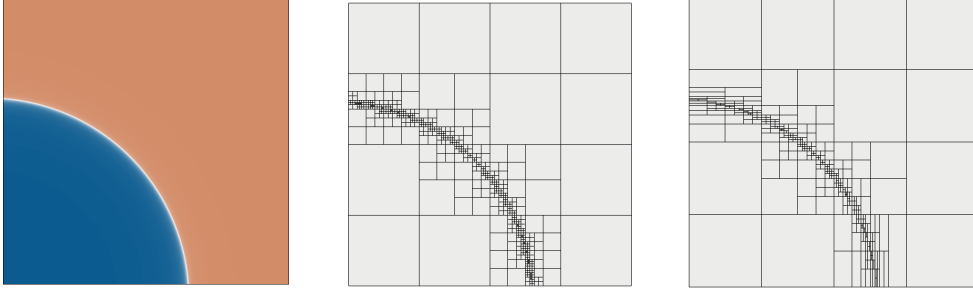


FIG. 7.1. 2D benchmark problem for finite element order $p=2$: solution (left), isotropic AMR mesh at 2197 DOFs (center), anisotropic AMR mesh at 1317 DOFs (right).

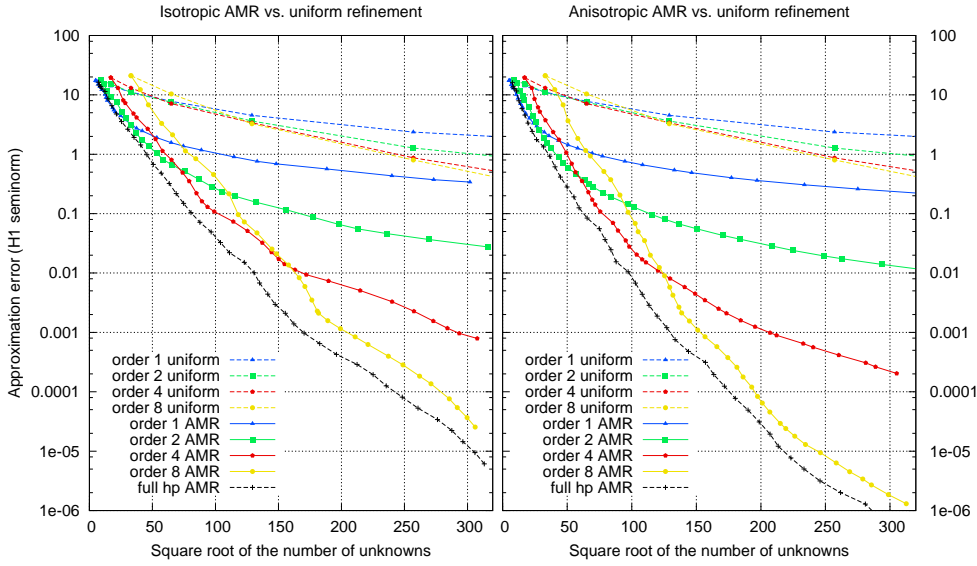


FIG. 7.2. Convergence history for the 2D benchmark: isotropic refinement (left) and anisotropic refinement (right). The horizontal axis shows the square root of the number of DOFs, to make it proportional to $1/h$ of the mesh.

in Step 4 above in order to avoid hanging nodes. As expected, the benefit of local refinement is substantial. Moreover, the benefit appears greater for higher order discretization. For completeness, we also include the best available result for the problem (according to [31], strategy REFSOLN_ELEM), obtained with hp -FEM, a method that adapts both the element size h and the element order p (the exact method used is described in [12]).

Anisotropic Refinement. Since our discretization supports anisotropic refinement, we test the same problem again and enable splitting quadrilaterals along one axis only, depending on the shape of the error. Assuming element K_i is selected for refinement in Step 4 based on its error e_i , we project and scale the error gradient along each transformed reference space axis of the element, and integrate the following anisotropic

error indicators:

$$a_j = \int_{K_i} \left(J^{(j)} \cdot \nabla(u_h - u) \right)^2 d\mathbf{x}, \quad (7.3)$$

where $J^{(j)}$ denotes the j -th column of the element transformation Jacobian. We then define the threshold $\tau_a = 0.6/dim \sum a_j$, and mark K_i for refinement in its j -th reference axis if $a_j > \tau_a$. Figure 7.1 (right) shows the anisotropically refined mesh after 12 iterations, and the convergence is compared to both isotropic and uniform refinement in Figure 7.2 (right). Even though the wave front in the solution is not really aligned with the mesh, many elements could still be refined in one direction only, which saved up to 48% DOFs in this problem.

3D Benchmark. Since our focus is 3D AMR, we present a straightforward generalization of the benchmark problem into 3D. This time, the right hand side is designed for the exact solution (shown in Figure 7.3, left) to be

$$u(x, y, z) = \arctan \left(\alpha \left(\sqrt{(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2} - r \right) \right), \quad (7.4)$$

with similar parameters, $\alpha = 200$, $(x_c, y_c, z_c) = (-0.05, -0.05, -0.05)$ and $r = 0.7$. We start with a mesh of $4 \times 4 \times 4$ hexahedra of degree p and execute the same AMR loop as in 2D, for both isotropic and anisotropic refinement.

Figure 7.3 shows the isotropic and anisotropic meshes after 12 and 14 iterations, respectively. Figure 7.4 plots the convergence histories of uniform refinement, isotropic AMR and anisotropic AMR. We obtain very similar behavior as in the 2D case. The convergence curves are smooth and do not reveal any irregularities. The 3D version of the problem benefits even more from anisotropic refinement: more than 80% DOFs are saved for $p=4$. Even a slight anisotropy in the solution means that elements with edge length ratios of 1:2, 1:4, or more, can be used.

Curved Meshes. One of our target applications is high-order Lagrangian hydrodynamics, where it is routine to use high-order (curvilinear) meshes. In MFEM, the mesh curvature is handled simply by maintaining a vector-valued finite element function that represents the physical position of all nodes and, in turn, of any point within the mesh. The associated finite element space is a subspace of $(H^1)^{dim}$, has its own conforming prolongation matrix to keep the curvature continuous, and does not need to coincide with the solution space. When the mesh is refined, the curvature function is interpolated from the original space to the finer space. We use a version of the anisotropic 3D benchmark problem to test this functionality on a spherical domain, as illustrated in Figure 7.5.

7.2. Parallel scalability. To demonstrate the parallel scalability of our algorithms, we designed a test to put as much stress on the AMR infrastructure as possible. We choose the simple Poisson problem again and fix the element order to $p = 2$. This is the lowest order that has both edge and face DOFs and where the stiffness matrix is still cheap to assemble. We also omit the linear solver and instead only perform nodal interpolation of the exact solution. This exposes the following components of the AMR iteration:

- constructing the P matrix,
- assembling the parallel AMR system (using *hypre*'s *RAP* function),
- refining elements and synchronizing ghost layers,
- load balancing the whole mesh at the end of the iteration.

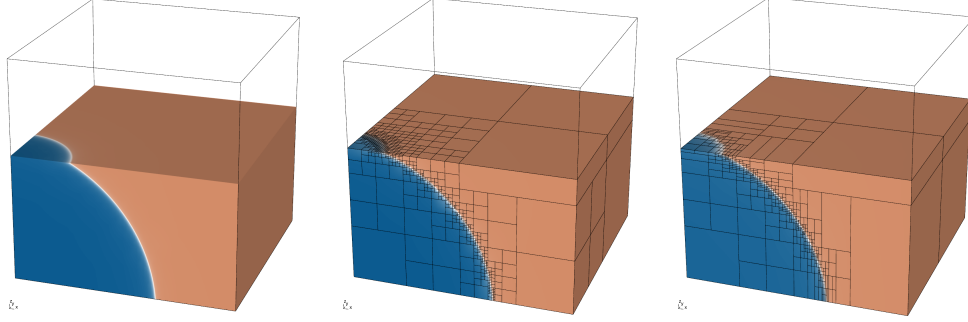


FIG. 7.3. 3D benchmark problem for order $p=1$: solution (left), isotropic AMR mesh at 12303 DOFs, error 5.142 (center), anisotropic AMR mesh at 5091 DOFs, error 4.999 (right).

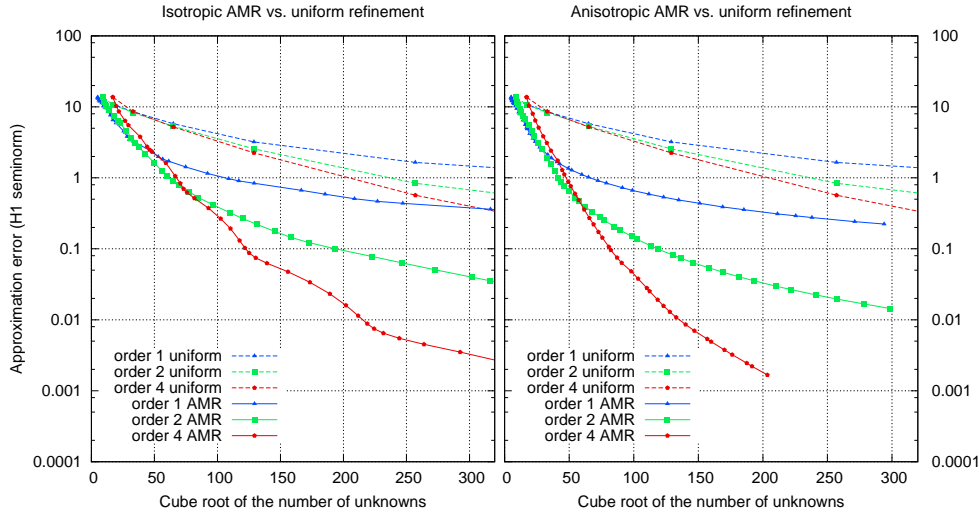


FIG. 7.4. Convergence history for the 3D benchmark: isotropic refinement (left) and anisotropic refinement (right). The horizontal axis is again normalized to correspond to average $1/h$ of the mesh.

For the exact solution, we reuse the 3D “wave front” function from the previous section. To make the problem larger, we sum two of the functions with radii 0.2 and 0.4, and center both at $(x_c, y_c, z_c) = (0.5, 0.5, 0.5)$. The gradient is also steeper with $\alpha = 400$. We initialize the mesh to 32^3 hexahedra and repeat the following steps, measuring their wall clock times (averaged over all MPI ranks):

1. Construct the finite element space for the current mesh (create the P matrix).
2. Assemble locally the stiffness matrix A and right hand side b .
3. Form the products $P^T A P$, $P^T b$.
4. Eliminate Dirichlet boundary conditions from the parallel system.
5. Project the exact solution u to u_h by nodal interpolation.
6. Integrate the exact error $e_i = \|u_h - u\|_{E, K_i}$ on each element.
7. Refine elements with $e_j > 0.9 \max\{e_i\}$. Synchronize ghost layers.
8. Load balance so each process has the same number of elements (± 1): redistribute elements and synchronize ghost layers again.

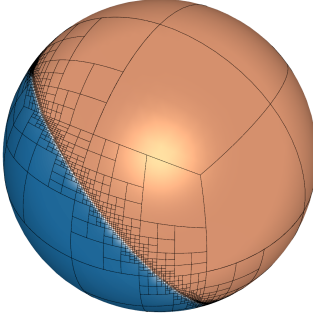


FIG. 7.5. AMR test on a spherical domain approximated by a degree 4 curvature function.

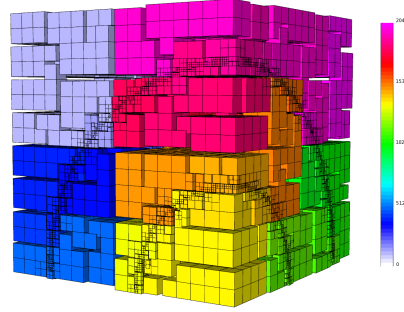


FIG. 7.6. One octant of the parallel scaling test mesh. Partitioning by the Hilbert curve is illustrated (2048 domains shown).

Figure 7.6 shows 1/8 of the mesh after several iterations of this AMR loop on 16384 MPI tasks. The colors represent MPI rank assignment.

Out of the approximately 100 iterations of the AMR loop, we select 8 iterations that have approximately 0.5, 1, 2, 4, 8, 16, 32 and 64 million elements in the mesh at the beginning of the iteration and plot their times as if they were 8 independent problems. This is possible because the sequence of meshes is always the same, regardless of the number of CPU cores used. We run from 64 to 393216 (384K) cores on LLNL's Vulcan BG/Q machine. Starting with 65536 cores, there are fewer elements in the initial mesh than parallel tasks, but this is not a problem in our implementation.

Figure 7.7 shows the total times for the selected iterations. The solid lines show strong scaling, i.e. we follow the same AMR iteration and its total time as the number of cores doubles. The dashed lines skip to a twice larger problem when doubling the number of cores, showing weak scaling, which should ideally be horizontal.

Figure 7.8 is a partial break-down of the total iteration time, showing individual scaling of dominant components of the iteration: P matrix construction, local assembly, RAP triple product, element refinement and load balancing. Local assembly scales perfectly because it contains no communication and the mesh is always perfectly balanced. The remaining steps do communicate, and so scale with different degrees of success, the worst being refinement and load balancing. Fortunately, these steps take relatively little time compared to the total iteration time, even in this synthetic benchmark containing no physics and solvers. Moreover, the load balancing time should be well worth the perfect scaling of the no-communication parts.

The last graph in Figure 7.8 shows the weak scaling of two memory measurements. The first is the maximum RSS (resident set size) obtained for each MPI process with `getrusage()`. This is the high watermark of the RAM allocated for the process by the operating system. The second measurement is the total memory (in MB) used by the non-conforming mesh data structure only. This is the more difficult of the two tests, as this particular data structure stores the ghost elements and other support data that the rest of the code does not see and that could potentially scale badly. For both measurements, the maximum over all MPI ranks is shown. The four curves in each set correspond to the top four weak scaling curves in Figure 7.7. The remaining weak scaling curves, as well as the strong scaling curves, are not shown here.

Finally, to test support for more than 2^{32} DOFs, we let the problem run on 128K

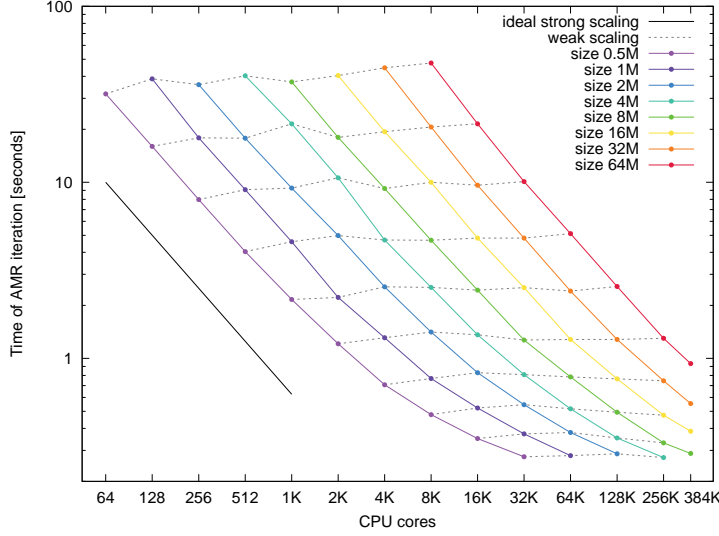


FIG. 7.7. Overall parallel scaling for selected iterations of the AMR loop.

cores with no limit on the number of iterations, until the available RAM was exhausted (1GB per process in this case). Execution stopped after 131 iterations when the mesh contained $1.94 \cdot 10^9$ elements, corresponding to $14.8 \cdot 10^9$ unknowns.

7.3. Dynamic AMR in Lagrangian hydrodynamics code. In this section we show that variational restriction-based AMR can be remarkably unintrusive when it comes to integration in a real finite element application code. For a demonstration, we chose the open-source high-order hydrodynamics solver Laghos [26], which simulates the time-dependent Euler equations of compressible gas dynamics in a Lagrangian frame, using high-order finite element spatial discretization and explicit high-order time-stepping. The full numerical method for Laghos is described in [16]. Here we only briefly mention that the simulation state consists of material position $x(t)$, velocity field $v(t)$ and internal energy $e(t)$ defined over the initial domain $\Omega(t_0)$. Kinematic quantities (position, velocity) are discretized by order p continuous elements, and thermodynamic quantities (energy) are discretized by order $(p-1)$ discontinuous elements. The discretized material position $x(t)$ corresponds directly to the mesh curvature function, i.e., the mesh moves and deforms together with the material. The gas density $\rho(t)$ can be calculated at any time from the deformation $x(t)$ and is not part of the simulation state.

To obtain the change in velocity at each time step, Laghos solves the linear system

$$M \frac{dv}{dt} = f, \quad (7.5)$$

where M is the kinematic mass matrix and f contains forces that depend on $x(t)$, $e(t)$ and the equation of state. To handle domain decomposition in a parallel computation, the system solved is really

$$P^T M P y = P^T f, \quad \frac{dv}{dt} = P y, \quad (7.6)$$

where the prolongation matrix P assembles contributions from shared degrees of freedom on parallel task boundaries (note that the product $P^T M P$ may never get

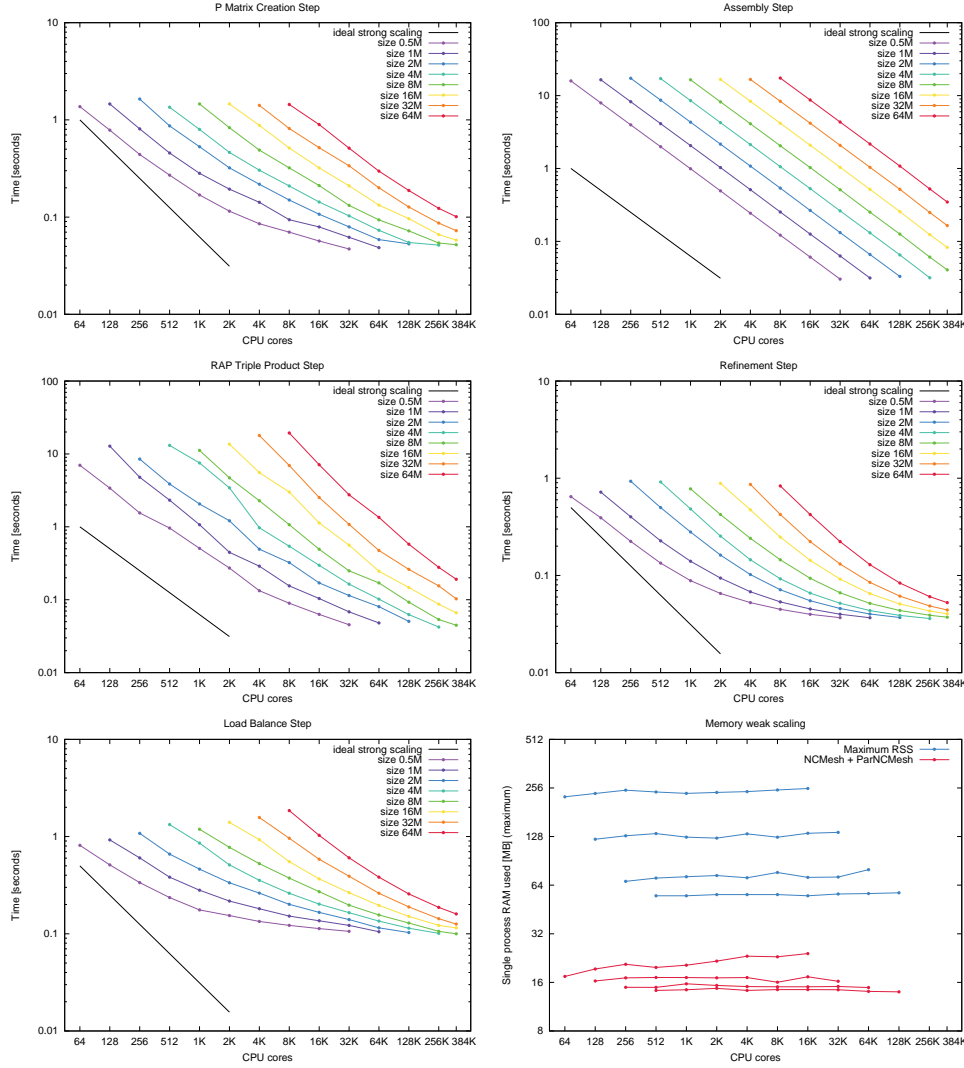


FIG. 7.8. Scaling of individual AMR loop steps, and memory weak scaling. Top row: P matrix construction, local assembly. Middle row: RAP triple product step, element refinement. Bottom row: load balancing step, memory weak scaling.

evaluated explicitly in the actual implementation). Since Laghos is already using variational restriction for domain decomposition, running the simulation on a parallel non-conforming mesh amounts to switching the P matrix to the one constructed in Section 6.5. The only minor issue is the order of elimination of essential boundary conditions, as discussed at the end of Section 3.

Our first experiment is running the 2D multi-material shock triple-point test problem (also described in [16]) on a static non-conforming mesh, meaning the initial mesh is already refined and does not change during the simulation. Based on a previous run on a coarse mesh, we anisotropically refine elements in the three material subdomains of the initial mesh in an attempt to counter the deformations the elements will undergo, thus improving the size of the CFL-limited time step. The initial and

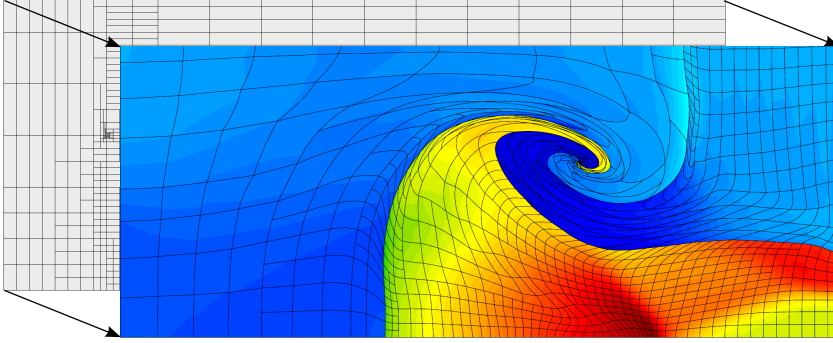


FIG. 7.9. *Static refinement in the triple-point problem. Initial mesh at $t = 0$ (background) refined anisotropically in order to obtain more regular element shapes at target time (foreground).*

target meshes are shown in Figure 7.9. We observe that some element deformations (diagonal compression) cannot be countered in this way, but we conclude that the code is capable of running on an arbitrarily refined non-conforming mesh practically out-of-the-box.

For the second experiment, we modified the Laghos code to allow for dynamic AMR that changes the mesh during the simulation, both via refinement and derefinement. Changing the mesh entails interpolating the state functions on the new mesh and reassembling both the kinematic and thermodynamic mass matrices (which are normally constant). After each mesh change we also load balance the mesh immediately (see Section 6.3). The state vectors are migrated together with the mesh using a parallel Boolean matrix constructed for each finite element space as part of the load balancing operation.

Since devising a general AMR strategy for Lagrangian shock hydrodynamics is beyond the scope of this paper, for the purpose of this demonstration we employ simple refinement and derefinement rules tailored specifically for the well known Sedov blast problem [16, 34]. Specifically, we look at the artificial viscosity coefficient to detect the presence of shock discontinuity and trigger refinement to a predefined maximum level in the undisturbed region of the mesh ($v = 0$) in front of the shock. In the post-shock region (nonzero v) we derefine a group of eight elements whenever their maximum densities are below a threshold defined as $\tau = 0.7 \max\{\rho(x, t)\}$, $x \in \Omega$. The initial mesh is the unit cube refined 5 times towards the origin, i.e., the corner where all internal energy is deposited at time $t = 0$. This corner is never derefined. For better stability, we also enforce 1-irregularity of the mesh at all times.

With these rules we obtain a dynamic AMR computation where the maximum refinement follows the shock, but the remaining areas are resolved with larger elements (see Figure 7.10). This AMR-enabled version of Laghos is publicly available in [26] and contains about 550 new lines of code. In contrast, obtaining these types of AMR results in similar codes has required multiple man-years of software development in the past.

8. Conclusions and future work. In this paper we presented a highly scalable approach to unstructured non-conforming adaptive mesh refinement that is easy to integrate in applications. The variational restriction approach clearly separates the finite element assembly from the influence of the non-conforming space. The conforming interpolation algorithm handles a large class of triangular, quadrilateral, prismatic

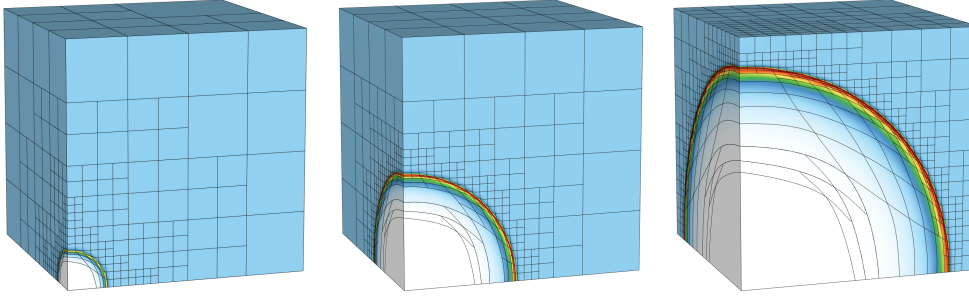


FIG. 7.10. *Dynamic refinement/derefinement in the 3D Sedov blast problem. Mesh and density shown at $t = 0.0072$ (left), $t = 0.092$ (center) and $t = 0.48$ (right). $Q3Q2$ elements ($p = 3$ kinematic, $p = 2$ thermodynamic quantities).*

and hexahedral meshes and can be used to construct arbitrary high order finite element discretizations in H^1 , $H(\text{curl})$ and $H(\text{div})$, as demonstrated in examples in this paper and in the freely available implementation in the MFEM library [28].

Uniquely, our approach allows anisotropic refinement and supports arbitrarily irregular meshes, which to the best of our knowledge has not been done previously for continuous Galerkin hexahedral elements. Compared to purely octree-based algorithms, we sacrifice some memory and runtime efficiency for generality (e.g., octrees only admit isotropic refinement). In the future, we plan to extend the method to 4D hypercubes and perform independent space and time adaptation, for which anisotropic refinement is crucial. We see this work as a first step towards that goal.

Reproducibility. Implementation of the described methods is available as open source software [28]. The model problems in Sections 7.1 and 7.2 are similar to Example 6p in MFEM. The dynamic refinement Sedov blast problem is available verbatim in the `amr` subdirectory of the Laghos miniapp [26].

REFERENCES

- [1] M. AINSWORTH AND B. SENIOR, *Aspects of an adaptive hp-finite element method: Adaptive strategy, conforming approximation and efficient solvers*, Computer Methods in Applied Mechanics and Engineering, 150 (1997), pp. 65–87. [2](#)
- [2] I. BABUSKA AND W. RHEINBOLDT, *Reliable error estimation and mesh adaptation for the finite element method*, (1979), pp. 67–108. [7](#)
- [3] W. BANGERTH, C. BURSTEDDE, T. HEISTER, AND M. KRONBICHLER, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Trans. Math. Softw., 38 (2012), pp. 14:1–14:28. [14](#)
- [4] W. BANGERTH, R. HARTMANN, AND G. KANSCHAT, *deal.II—a general-purpose object-oriented finite element library*, ACM Trans. Math. Softw., 33 (2007). [2](#)
- [5] R. E. BANK, A. H. SHERMAN, AND A. WEISER, *Refinement algorithms and data structures for regular local mesh refinement*, Scientific Computing, (1983), pp. 3–17. [2](#)
- [6] E. G. BOMAN, U. V. CATALYUREK, C. CHEVALIER, AND K. D. DEVINE, *The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring*, Scientific Programming, 20 (2012), pp. 129–150. [12](#)
- [7] C. BURSTEDDE, L. C. WILCOX, AND O. GHATTAS, *p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM J. Sci. Comput., 33 (2011), pp. 1103–1133. [2](#), [8](#), [11](#), [12](#), [13](#)
- [8] P. M. CAMPBELL, K. D. DEVINE, J. E. FLAHERTY, L. G. GERVASIO, AND J. D. TERESCO, *Dynamic octree load balancing using space-filling curves*, Tech. Rep. CS-03-01, Williams College Department of Computer Science, 2003. [12](#), [13](#)

- [9] G. F. CAREY, *A mesh-refinement scheme for finite element computations*, Computer Methods in Applied Mechanics and Engineering, 7 (1976), pp. 93 – 105. [2](#)
- [10] C. CARSTENSEN AND J. HU, *Hanging nodes in the unifying theory of a posteriori finite element error control*, J. Comput. Math., 27 (2009), pp. 215–236. [7](#)
- [11] *CEED: Center for Efficient Exascale Discretizations*, Exascale Computing Project, DOE. <http://ceed.exascaleproject.org>. [1](#)
- [12] J. ČERVENÝ, *Automatic hp-Adaptivity for Time-Dependent Problems*, PhD thesis, University of West Bohemia, 2012. [18](#)
- [13] P. G. CIARLET, *The Finite Element Method for Elliptic Problems*, North-Holland, 1978. [3](#)
- [14] L. DEMKOWICZ, J. ODEN, W. RACHOWICZ, AND O. HARDY, *Toward a universal h-p adaptive finite element strategy, part 1. constrained approximation and data structure*, Computer Methods in Applied Mechanics and Engineering, 77 (1989), pp. 79 – 112. [1](#), [2](#), [7](#)
- [15] V. DOBREV, T. KOLEV, C. LEE, V. TOMOV, AND P. VASSILEVSKI, *Algebraic hybridization and static condensation with application to scalable $H(\text{div})$ preconditioning*, SIAM Journal on Scientific Computing, to appear (2019). [2](#), [5](#)
- [16] V. A. DOBREV, T. V. KOLEV, AND R. N. RIEBEN, *High-order curvilinear finite element methods for Lagrangian hydrodynamics*, SIAM Journal on Scientific Computing, 34 (2012), pp. B606–B641. [22](#), [23](#), [24](#)
- [17] J. FLAHERTY, R. LOY, M. SHEPHARD, B. SZYMANSKI, J. TERESCO, AND L. ZIANTZ, *Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws*, J. Parallel Distrib. Comput., 47 (1997), pp. 139–152. [2](#)
- [18] M. GRIEBEL AND G. ZUMBUSCH, *Parallel multigrid in an adaptive pde solver based on hashing and space-filling curves*, 1997. [12](#)
- [19] V. HEUVELINE AND F. SCHIEWECK, *H1-interpolation on quadrilateral and hexahedral meshes with hanging nodes*, Computing, 80 (2007), pp. 203–220. [1](#)
- [20] *hypr: A library of high performance preconditioners*. <http://www.llnl.gov/CASC/hypr/>. [12](#)
- [21] T. ISAAC, C. BURSTEDDE, L. WILCOX, AND O. GHATTAS, *Recursive algorithms for distributed forests of octrees*, SIAM Journal on Scientific Computing, 37 (2015), pp. C497–C531. [2](#)
- [22] T. ISAAC AND M. G. KNEPLEY, *Support for non-conformal meshes in PETSc's DMplex interface*, CoRR, abs/1508.02470 (2015). [8](#)
- [23] G. KARYPIS AND V. KUMAR, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, J. Parallel Distrib. Comput., 48 (1998), pp. 71–95. [2](#)
- [24] B. S. KIRK, J. W. PETERSON, R. H. STOGNER, AND G. F. CAREY, *libMesh: A C++ library for parallel adaptive mesh refinement/coarsening simulations*, Eng. with Comput., 22 (2006), pp. 237–254. [2](#)
- [25] M. G. KNEPLEY AND D. A. KARPEEV, *Mesh algorithms for PDE with Sieve I: mesh distribution*, Sci. Program., 17 (2009), pp. 215–230. [8](#)
- [26] *Laghos: High-order Lagrangian hydrodynamics miniapp*. <http://github.com/CEED/Laghos>. [22](#), [24](#), [25](#)
- [27] P. LINDSTROM, *The minimum edge product linear ordering problem*, Tech. Rep. LLNL-TR-496076, Lawrence Livermore National Laboratory, Aug. 2011. [12](#)
- [28] *MFEM: Modular finite element methods*. <http://mfem.org>. [12](#), [13](#), [17](#), [25](#)
- [29] W. F. MITCHELL, *Hamiltonian paths through two- and three-dimensional grids*, J. Res. Natl. Inst. Stand. Technol., 110 (2005), pp. 127–36. [2](#)
- [30] W. F. MITCHELL, *A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids*, J. Parallel Distrib. Comput., 67 (2007), pp. 417–429. [2](#), [12](#), [13](#)
- [31] W. F. MITCHELL AND M. A. MCCLAIN, *A Survey of hp-Adaptive Strategies for Elliptic Partial Differential Equations*, Springer Netherlands, Dordrecht, 2011, pp. 227–258. [17](#), [18](#)
- [32] R. SCHNEIDERS, *Refining quadrilateral and hexahedral element meshes*, Fifth International Meshing Roundtable, 1 (1998). [1](#)
- [33] T. SCHÖNFELD, *Adaptive mesh refinement methods for three-dimensional inviscid flow problems*, International Journal of Computational Fluid Dynamics, 4 (1995), pp. 363–391. [1](#)
- [34] L. I. SEDOV, *Similarity and Dimensional Methods in Mechanics*, CRC Press, 10th ed., 1993. [24](#)
- [35] F. T. SUTTMEIER, *On concepts of PDE-software: The cellwise oriented approach in DEAL*, International Mathematical Forum, 2 (2007), pp. 1 – 20. [2](#)
- [36] T. TU, D. O'HALLARON, AND O. GHATTAS, *Scalable parallel octree meshing for terascale applications*, in Proc. of the 2005 ACM/IEEE Conf. on Supercomputing, 2005, pp. 4–. [12](#)
- [37] P. ŠOLÍN, *Partial Differential Equations and the Finite Element Method*, John Wiley & Sons, 2005. [4](#)
- [38] P. ŠOLÍN, J. ČERVENÝ, AND I. DOLEŽEL, *Arbitrary-level hanging nodes and automatic adaptivity in the hp-fem*, Math. Comput. Simul., 77 (2008), pp. 117–132. [2](#)

- [39] E. WILSON, *The static condensation algorithm*, Int. J. Numer. Methods Eng., 8 (1974), pp. 199–203. 5