Engineering With Computers manuscript No.
(will be inserted by the editor)

`libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations

Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, Graham F. Carey

CFDLab
The University of Texas at Austin
Dept. of Aerospace Eng. and Eng. Mechanics
1 University Station C0600
Austin, TX 78712

Received: June 26, 2006/ Revised version: June 26, 2006

Abstract In this paper we describe the `libMesh` framework for parallel adaptive finite element applications. `libMesh` is an open-source software library that has been developed to facilitate serial and parallel simulation of multiscale, multiphysics applications using adaptive mesh refinement and coarsening strategies. The main software development is being carried out in the CFDLab at the University of Texas, but as with other open-source software projects; contributions are being made elsewhere in the U.S. and abroad. The main goals of this article are: (1) to provide a basic reference source that describes `libMesh` and the underlying philosophy and software design approach; (2) to give sufficient detail and references on the adaptive mesh refinement and coarsening (AMR/C) scheme for applications analysts and developers; and (3) to describe the parallel implementation and data structures with supporting discussion of domain decomposition, message passing, and details related to dynamic repartitioning for parallel AMR/C. Other aspects related to C++ programming paradigms, reusability for diverse applications, adaptive modeling, physics-independent error indicators, and similar concepts are briefly discussed. Finally, results from some applications using the library are presented and areas of future research are discussed.

Key words parallel computing – adaptive mesh refinement – finite elements

1 Introduction

The `libMesh` library was created to facilitate parallel, adaptive, multiscale, multiphysics finite element simulations of increasing variety and difficulty in a reliable, reusable way. Its creation was made possible by two main factors: the first was the existence of a relatively robust and rapidly-evolving parallel hardware-software infrastructure that included affordable parallel distributed clusters running Linux and high performance implementations of the MPI standard [1]. The second was the evolution of adaptive mesh methodology, and algorithms for domain decomposition and efficient repartitioning.

A major goal of `libMesh` is to provide a research platform for parallel adaptive algorithms. Centralizing physics-independent technology (and leveraging existing libraries whenever possible) amortizes the formidable software development effort required to support parallel and adaptive unstructured mesh-based simulations. Users can then focus on the specifics of a given application without considering the additional complexities of parallel and adaptive computing. In this way `libMesh` has proved a valuable testbed for a wide range of physical applications (as discussed further in section 8).

The development of `libMesh` in the CFDLab research group was in some sense a natural consequence of: (1) a long history of developing methodology and software for simulations that involved unstructured adaptive mesh refinement [2]; (2) an early commitment to parallel computing and subsequently to building and using distributed parallel PC clusters [3]; (3) research experience in developing parallel libraries and software frameworks using advanced programming concepts and tools; and (4) an enthusiastic small group of researchers in the CFDLab committed to implement this design. In the ensuing discussion we elaborate on the parallel AMR/C design and the infrastructure, methodology, and software technology underlying `libMesh`.

The simulation methodology in `libMesh` employs a standard cell-based discretization discussed in many introductory texts on finite elements (c.f. [4]), using adaptive mesh refinement to produce efficient meshes which resolve small solution features (c.f. [5–7]). The adaptive technology utilizes element subdivision to locally refine the mesh and thereby resolve different scales such as boundary layers and interior shock layers [8]. Different finite element formulations may be applied including Galerkin, Petrov-Galerkin, and discontinuous Galerkin methods. Parallelism is achieved using domain

decomposition through mesh partitioning, in which each processor contains the global mesh but in general computes only on a particular subset. Parallel implicit linear systems are supported via an interface with the PETSc library. Since AMR/C is a dynamic process, efficient load balancing requires repartitioning in both steady-state and evolution problems. In evolution problems, in particular, coarsening is desirable and sometimes essential for obtaining efficient solutions. Later, we discuss the present approach for simultaneous refinement and coarsening for time-dependent applications. We also summarize the current state-of-the-art in *a posteriori* error estimation and computable error indicators to guide local refinement, and specifically consider the issue of physics independent indicators for libraries such as `libMesh`.

There is a key difference between an AMR/C *library* and an application-specific AMR/C *code*. While the library approach provides the flexibility of treating a diverse number of applications in a generic and reusable framework, the application-specific approach may allow a researcher to more easily implement specialized error indicators and refinement strategies which are only suitable for a specific application class. Later in this paper, we will discuss both the present flexible implementation and also mechanisms for using more sophisticated error indicators that are targeted to the applications.

The concept of separating to some extent the physics from the parallel AMR/C infrastructure was influenced by a prior NASA HPC project under which we developed a parallel multiphysics code for very large-scale applications [9]. Here we applied software design principles for frameworks [10], utilized formal revision control procedures, and developed rigorous code verification test suites. The parallel solution algorithms discussed here drew on earlier experience developing a prototype parallel Krylov solver library for a non-overlapping finite element domain decomposition [11].

In the following sections, we discuss in more detail the AMR/C background and ideas implemented in `libMesh`, the parallel domain decomposition approach including the data structures for element neighbor treatment, and the object-oriented design as well as the overall structure of `libMesh`. Numerical simulations and performance results for representative illustrative applications in two and three dimensions are included. In the concluding remarks, we discuss some of the open issues, limitations, and opportunities for future extensions of `libMesh`.

2 Adaptive Mesh Refinement and Coarsening (AMR/C)

As indicated in the introductory remarks, AMR/C has been a topic of research and application interest for some time. Perhaps the earliest studies were those for elliptic problems in 2D using linear triangular elements with hanging node constraints enforced explicitly at the interface edge between a refined and coarsened element [12, 13]. Physics-independent "solution feature" indicators were used in this 2D AMR research software, and physics-based residual indicators were used in [14]. Other work since that period includes the PLTMG [15] software for 2D meshes of triangles. In this code,

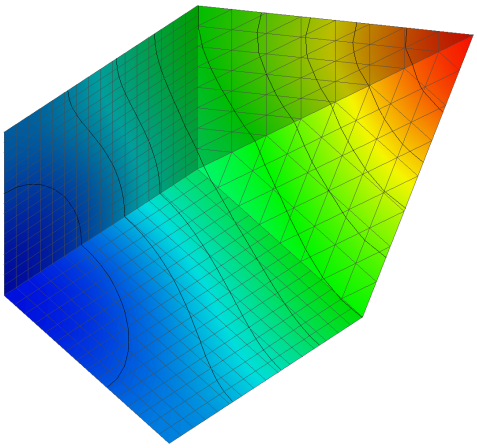

Fig. 1 h-refinement for a simple Helmholtz problem using a hybrid prism/tetrahedral mesh. Algebraic constraints enforce continuity across "hanging nodes"

"mesh conformity" is enforced by connecting a mid-edge node to the opposite vertex of a neighbor element. This has been extended by others to progressive longest-edge bisection of tetrahedral meshes [16, 17]. Constraints at mid-edge nodes can also be conveniently handled by algebraic techniques or multiplier methods [2, 18].

These AMR concepts have been extended to 3D and transient applications, and other AMR extensions such as refinement of local polynomial degree (p methods) as well as combined element subdivision and p-refinement (hp) methods. An example of h refinement in `libMesh` is given in Figure 1 for a hybrid mesh. The focus in `libMesh` is on local subdivision (h refinement) with local coarsening by h restitution of subelements. However, `libMesh` does permit h refinement with uniformly high degree elements, and the developmental branch of `libMesh` now supports adaptively p refined and hp refined meshes with some element types.

The error indicator work to date in `libMesh` has focused on local indicators that are essentially independent of the physics. This allows the library to be more flexibly applied in diverse applications. On the other hand, there is an extensive literature devoted to obtaining more reliable *a posteriori* estimates and accompanying error indicators that are more closely linked to the operators and governing equations for the application problem. Bounds relating the global error in energy to residuals with respect to the approximate solution and governing equations have been developed and their properties extensively studied. Refinement indicators based on local element residual contributions are a natural consequence and the most common form of AMR indicator in the literature. Recently, some more precise and reliable indicators that involve the additional work of solving a related dual problem have been proposed and are the subject of ongoing research [19–23].

Residual indicators and targeted dual indicators are difficult to include in a flexible library without compromising the goal of physics-independence. In the most general case,

an error estimator should be able to take only a finite element mesh and a function expressed on that mesh and return approximate error levels on every element. libMesh provides simple interface derivative jump (or flux jump) indicators, the latter only in the case where there is a discontinuity in material coefficient of the flux vector across the element edge [24]. These only give rigorous error bounds for a very limited class of problems, but, in practice, they have proved to be broadly applicable. Another class of physics-independent indicators that is very widely used because of their simplicity are the "recovery indicators". In this case the gradient or solution is recovered via local patch post-processing [25]. Superconvergence properties may be used to provide a more accurate post-processed result [26]. A more rigorous foundation for these indicators has been determined [27]. The difference between the more accurate recovered local value and the previously-computed value provides the local error indicator for the refinement process. The ability for error indicators to provide improved local solutions in addition to error estimates is crucial for automatic hp schemes, where in addition to choosing which elements to refine it is also necessary to choose how (h-subdivision or p-enrichment) to refine them.

There is a strong interest in utilizing parallel AMR/C in the setting of the SIERRA framework being developed at Sandia National Laboratory [28]. Recovery indicators are being implemented in SIERRA for many of the same reasons they are used in libMesh. Sandia researchers are also engaged in collaborations with University researchers to test and utilize the dual type indicators. To include such dual indicators in libMesh for a specific application we could add a residual calculation as a post-processing step and then again utilize libMesh for an approximate solution of the linearized dual problem in terms of the target quantity of interest using an "appropriate" mesh. An alternative model would be to solve the dual problem locally for an approximation to this error indicator contribution.

The coarsening aspect of AMR/C merits further discussion. Coarsening occurs whenever a "parent" element is reactivated due to the deactivation of all of its "child" elements. This scheme assumes the existence of an initially conforming mesh, whose elements are never coarsened. There are many practical issues related to the selection of elements for refinement and coarsening, notwithstanding the calculation of an accurate error indicator. Special rules such as "refine any element which is already coarser than a neighbor chosen for refinement" or "refine any element which would otherwise become coarser than all its neighbors" may be chosen to smooth the mesh grading and force additional refinement in regions with otherwise small error.

One of the approaches that we have been investigating is a statistical strategy for proportioning cells between refinement and coarsening. The ideas are related to earlier approaches [2] in which the mean μ and standard deviation σ of the indicator "population" are computed. Then, based on refinement and coarsening fractions r_f and c_f (either default values or specified by the user), the elements are flagged for refinement and coarsening. This scheme, depicted graphically in Figure 2, is beneficial in evolution problems where, at early times, the error is small and equidistributed and no elements are flagged for refinement. Later, as interesting features develop, the statistical distribution spreads and refinement and coarsening begins. As the steady solution is approached, the distribution of the error reaches a steady state as well, effectively stopping the AMR/C process.

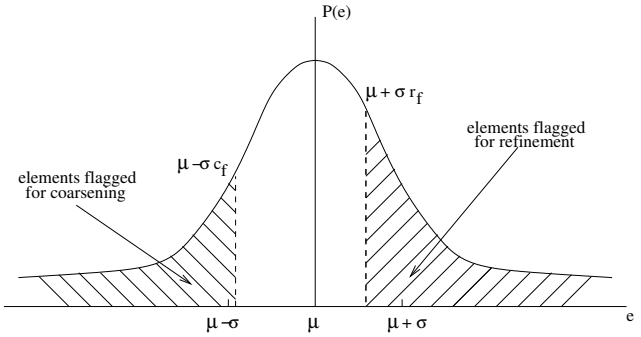

Fig. 2 In the statistical refinement scheme, the element error e is assumed to have an approximately normal probability density function $P(e)$ with mean μ and standard deviation σ. Elements whose error is larger than $\mu + \sigma r_f$ are flagged for refinement while those with errors less than $e < \mu - \sigma c_f$ are flagged for coarsening.

We remark that other standard strategies for refinement and coarsening are also used with libMesh. The optimal strategy for selecting elements for refinement is somewhat problem-dependent and is an area of future research. Some further discussion is presented in Section 8.

3 Library Overview

Experiences with a number of other libraries demonstrated the feasibility of developing high performance parallel numerical libraries in C++ [29–31] and influenced the libMesh design. As in deal.II, libMesh was designed from the beginning to use advanced features of the C++ programming language. No provision is made for lower-level procedural languages such as C or Fortran. This is in contrast to other parallel frameworks such as Cactus[1] or ParFUM[2]. However, exposing the class and template structure of libMesh to users can increase performance and facilitates extensibility, and both are features we value above inter-language operability. Some other high-performance library designs that have influenced libMesh are [32–34].

The libMesh project began in March 2002 with the goal of providing a parallel framework for adaptive finite element simulations on general unstructured meshes. The library is distributed under an open-source software license and hosted

[1] http://www.cactuscode.org
[2] http://charm.cs.uiuc.edu/research/ParFUM

by Sourceforge. Geographically dispersed development is managed with the Concurrent Versions System (CVS) software. The online documentation for the library has averaged approximately 40,000 hits a month since January 2005. The library itself has been downloaded on average approximately 150 times a month.

3.1 Scope

The library was originally intended to provide a powerful data structure which supports adaptive mesh refinement for arbitrary unstructured meshes arising in finite element and finite volume simulations. By separating the adaptive meshing technology from the application code, the potential for code reuse increases dramatically, and this is evident from the growing number of diverse applications which now exploit the library. Some application results are presented in Section 8.

Subsequent development efforts have been targeted at increasing performance, supporting more general classes of finite elements, and implementing specific solution algorithms for transient and nonlinear problems. A major goal of the library is to provide support for adaptive mesh refinement computations in parallel while allowing a research scientist to focus on the physics being modeled. To this end the library attempts to hide complications introduced by parallel computing whenever possible so that the user can focus on the specifics of the application.

Both AMR and parallelism offer means to accelerate simulation and analysis for design and rapid prototyping: parallel speed-up reduces the real time to solution and likewise AMR permits a solution to be achieved to comparable accuracy on a coarser but better designed mesh than with standard non-adaptive meshing. The goal in combining adaptivity and parallelism is clearly to reap the benefits of both techniques in being able to solve problems more efficiently and in shorter real time or, alternatively, being able to consider more complicated problems with a fixed set of resources. Of course, when one utilizes AMR or parallelism, additional layers of complexity are being added to the analysis problem, methodology, algorithms, data structures and software. There is also an overhead associated with the implementation of both AMR and parallelism and these factors should also be taken into account. Nevertheless, it is clear that each of these strategies offers the ability to greatly enhance the computational capability available to a researcher.

3.2 C++ and Scientific Computing

The library is written in C++, with code designed for the ISO standard but tested and restricted for compatibility with older Intel, IBM, GNU, and other compilers. The library uses polymorphism to enable a separation between application physics and finite element implementation. The support for object-oriented programming in C++ allows application authors to write their code around abstract class interfaces like `FEBase`, `QBase`, and `NumericVector`, then to switch at compile time or run time between different finite element types, quadrature rules, and linear solver packages which implement those interfaces. To reduce the overhead of virtual function calls to abstract base classes, we provide methods which encourage developers to use a few calls to large functions rather than many calls to small functions. For example, sparse matrices are constructed with one function call per element to add that element's cell matrix, rather than directly calling virtual `SparseMatrix` access functions for each degree of freedom pair at each quadrature point. For frequently called functions which cannot be combined in this way, `libMesh` uses C++ templates. We mimic the support for generic programming in the C++ Standard Template Library. For example, `libMesh` iterator classes make it easy for users to traverse important subsets of the elements and nodes contained in a mesh.

Our decision to use C++ is much in the spirit of Winston Churchill's famous opinion of democracy: "It is the worst system, except for all the others." This philosophy, advanced by the `Alegra` [30] developers in the mid 1990s, is still germane today. Although writing efficient C++ code can be difficult, the C++ language supports many programming styles, making it possible to write software with layers of complexity which are more easily maintainable than in "lower level" languages (e.g. C, Fortran) but with time-critical routines that have been more aggressively optimized than is possible in "higher level" languages. Fortran and C are fast and popular languages for numerical analysis, but do not adequately support the object-oriented methodology we wanted for `libMesh`. Java implements run time polymorphism via inheritance, but lacks the compile time polymorphism that C++ templates provide to produce faster executables. Java also lacks the operator overloading that `libMesh` numeric classes like `VectorValue` and `TensorValue` use to make formulas look more natural to a mathematician.

The existence of high-quality standards-conforming C++ compilers from hardware vendors, software vendors, and the GNU project helps keep `libMesh` portable to many different hardware platforms. The ease of linking C, Fortran, and assembly code into a C++ application also allows `libMesh` to make use of existing libraries written in lower-level languages. Also, C++ encapsulation provides a natural mechanism for interfacing with separate third-party libraries through a common interface (as discussed in Section 3.4), and object-oriented design is well-suited for handling the layers of complexity introduced by combined adaptivity and parallelism. Finally, C++ ranks with C and Java as one of the most popular programming languages among software developers, which has helped attract more end users to the `libMesh` library and more external contributors to `libMesh` development.

3.3 Open Source Software Development

The library and source code are distributed under the GNU Lesser General Public License (LGPL) [35]. The LGPL provides the benefits of an open source license but also allows the

library to be used by closed source and commercial software. This is important for the research community, because it allows applications using the library to be redistributed regardless of the application's license. In January 2003, the popular Sourceforge site was chosen to host the first official software release. Sourceforge provides services to aid software development including CVS repository management, web[3] and database hosting, mailing lists, and access to development platforms.

The CVS branch hosted at Sourceforge is frequently updated by the core group of authorized developers. Periodically the developmental branch is "frozen" in order to create an official release. During each freeze, major API changes are deferred while outstanding problems in the code are fixed and the library is tested on each supported platform to ensure portability. The public has CVS read access, so users can periodically modify their application codes to take advantage of new library features, or simply tie their application to a particular `libMesh` version.

Developers are sometimes recruited from the user community, when a user desires a specific library feature and, with help from the other developers, submits the new functionality as a patch to the current CVS tree. After testing the patch, an authorized developer can check it in to the active branch. Users who want to make significant, continuous improvements to the library are added as active developers and given CVS write access.

Accurate documentation is critical for the success of any C++ class library. In `libMesh` the well-known `doxygen` utility is used to extract documentation directly from source code [36]. This approach has the benefit of keeping the source code and its documentation synchronized. `doxygen` extracts blocks of comments and creates a well-organized web page which contains class documentation, detailed inheritance diagrams, and annotated source code.

3.4 Interfaces to Other Libraries

There are a number of existing, high-quality software libraries that address some of the needs of a simulation framework. In `libMesh`, we utilize existing software libraries whenever possible. It is crucial for a small development team to avoid the "not invented here" mindset, so that efforts may be focused as narrowly and effectively as possible. The most general support for third party software such as the hex mesh generator CUBIT [37] is provided through the mesh file format support discussed in Section 5.1, but `libMesh` can also be configured to directly link to supporting software when convenient. Some third party libraries in addition to the ones discussed below include the `boost` C++ source libraries, the 2D Delaunay triangulator `Triangle` [38], and the 3D tetrahedral mesh generator `tetgen` [39].

The library uses both METIS [40] and ParMETIS [41] for domain decomposition (discussed further in Section 4). The Zoltan library from Sandia National Labs provides a uniform interface to a number of mesh partitioning schemes [42] and would be natural to include in the future. Additional partitioning schemes can be added to the library very easily through the standard C++ approach of subclassing. The library provides the abstract `Partitioner` base class that defines the partitioning interface, and derived classes can serve as wrappers for external partitioning libraries.

The base class/derived class paradigm is also used to interface with third party linear algebra packages. In this case the library provides the abstract `SparseMatrix`, `NumericVector`, `LinearSolver`, and `EigenSolver` classes. Derived classes are then used to provide the actual implementation. This approach has been used to encapsulate the interface to solver packages such as LASPack [43], which provides Krylov subspace linear solvers for serial machines, PETSc, the parallel scientific computing toolkit from Argonne National Labs [44], and SLEPc, the library for eigenvalue problem computations from Universidad Politecnica de Valencia [45].

3.5 Portability

Portability across a number of platforms using native compilers has been a goal of the library design since its inception. The bulk of the development work is performed on Linux desktop machines using the GNU Compiler Collection, but a number of other platforms are supported as well. The library makes extensive use of the C++ Standard Template Library, so it is essential to use multiple compilers to ensure compiler-specific constructs are avoided.

The GNU `autoconf` package is used to configure the library for a given installation. This approach uses the familiar `configure` script to probe a user's computing environment for parameters such as compiler and external library versions. The configuration process also sets global options such as whether real or complex-valued scalars are to be used. This procedure produces a custom `Makefile` with site-specific information, and the library is built with GNU `make` or the vendor equivalent.

The desire to use native compilers is primarily performance driven. On architectures such as the IBM Power 5 and the Intel Itanium® II there are a number of complex instructions available, and vendor-supplied compilers seem to optimize well for these features. Additionally, when a new platform becomes available it is often vendor-supplied compilers which are available first. For these reasons the library has always been tested with a range of compilers before each official release. A side effect of this approach is that the library has subsequently been ported to additional architectures such as OSX and Windows with little difficulty.

One ongoing issue is portability across different versions of external libraries such as PETSc. The PETSc API often changes between minor releases, rendering code that was correct for one version inoperable with another. GNU `autoconf` and the C preprocessor are used to provide the correct code for the installed version of PETSc, and modifications are inevitably required with each subsequent PETSc release. At

[3] `http://libmesh.sourceforge.net`

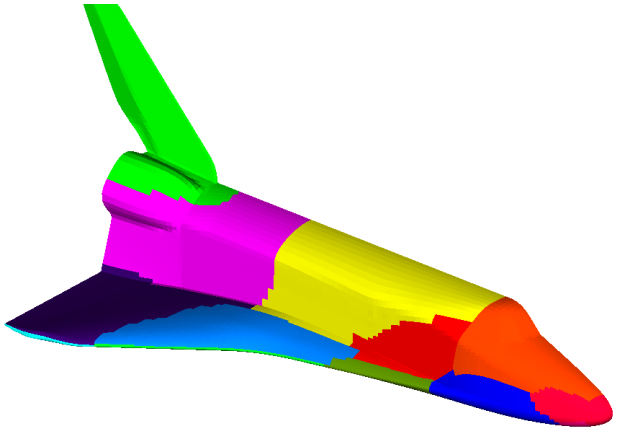

Fig. 3 Element-based domain decomposition of a surface mesh into 16 subdomains

the time of this writing, `libMesh` supports all versions of PETSc from 2.1.0 to 2.3.0. One way of getting around such issues is to actually distribute the source code for external libraries with `libMesh`. This approach is used for LASPack and `tetgen`, but is impractical for PETSc due to its size and build complexities.

4 Domain Decomposition

A standard non-overlapping domain decomposition approach is used in `libMesh` to achieve data distribution on parallel computers as shown in Figure 3 [2]. The discrete domain Ω_h is partitioned into a collection of subdomains: $\{\Omega_h^p\}$ such that $\bigcup \Omega_h^p = \Omega_h$ and $\bigcap \Omega_h^p = \emptyset$. The elements in each subdomain are assigned to an individual processor. The two primary metrics in judging the quality of a partition are the subdomain mesh size and the number of "edge cuts" in the resulting partition. For a mesh composed of a single type of element, each subdomain should contain an equal number of elements so that the resulting domain decomposition is load balanced across all available processors. The edge cut metric, on the other hand, is designed to minimize the interprocessor communication required by the parallel solver. For an overview of several domain decomposition strategies which are available, see [42, 46].

In problems with high-resolution static meshes, the partitioning is only performed once. In such cases, a high-quality partition which simultaneously minimizes both the size and edge-cut metrics may be desirable even though it is relatively expensive. For AMR/C applications where the steady-state solution is of interest, it is frequently the case that one begins with a coarse mesh at the root level and progressively refines towards a near-optimal mesh with little coarsening. It is obvious that an initially balanced partition may rapidly become very unbalanced here and lead to computational inefficiencies. Consequently, the mesh typically requires frequent repartitioning during the AMR process. The development of optimal schemes for repartitioning that can take advantage of a prior partition in a parallel AMR setting is still an open research issue [46].

In `libMesh` we partition by default with the recursive scheme provided by METIS when the number of selected partitions $n_p \leq 8$, and with the k-way scheme otherwise. A space filling curve partitioning algorithm is also available, as is an interface to ParMETIS. The frequency of repartitioning needed will in general depend on the evolving imbalance, and can occur as often as every time the mesh changes (i.e. every time refinement or coarsening occurs). Profiling suggests that this technique is not overly inefficient for typical applications, but it could be very slow for large-scale problems, and clearly it is unnecessary if the refinement scheme selects only a small number of elements to be refined and coarsened. This is one of many aspects of algorithmic performance which must be considered for a given application on a given computer platform. For further discussion see Section 8.

Another issue that must be considered is the subset of the AMR tree on which the partitioning algorithm acts. Typically, the partitioning algorithm is applied to all the active elements (i.e., the leaves of the AMR tree) so that subsequent calls to the matrix assembly routine can be effectively parallelized. However, this may involve calling the partitioning algorithm on a large subset of the AMR tree when it may be sufficient to partition based on a coarser level and simply assign all the children of these coarse level elements to the same processor. Due to the parallel implementation of the `Mesh` discussed in Section 7, we do not (yet) consider the possibility that accessing an ancestor element would require off-processor communication. In such a scenario, one would need to ensure that repeated refinement and coarsening of the same element did not lead to excessive communication overhead, perhaps by ensuring that a local synchronized copy of an element's parent is always available.

5 Data Structures

This section describes several of the key data structures in `libMesh`. The discussion focuses on basic functionality, possible extensions, and the reasoning behind certain design decisions. Algorithms that are central to the library's functionality are also described.

5.1 Mesh

The `Mesh` class is central to `libMesh` and was one of the first developed. It provides a discrete description of an object in d-dimensional space, where d is 1, 2, or 3. The discretization is composed of elements and nodes which are stored in the mesh, but the manner in which these data are stored is encapsulated by abstract classes with implementation-independent interfaces. This data encapsulation has allowed for refactoring of the mesh class with minimal impact on the external application programming interface.

A base class/derived class structure is used to implement mesh I/O in various formats. Virtual base classes describe the interface for mesh input and output, and derived classes provide the actual I/O functionality. The library supports reading and writing a number of unstructured mesh formats, including the UCD format from AVS, the I-deas Universal format UNV, Exodus II from Sandia National Labs, GMSH, TetGen, Tecplot (ASCII and binary) and GMV from Los Alamos National Labs. The initial mesh is assumed to be conforming and provides the level-0 parent elements in the refinement hierarchy described in section 5.4.3.

Custom iterator objects can be created to provide access to the elements and nodes contained in a mesh. The user can instantiate iterators to access all the elements in the mesh or some meaningful subset thereof. The latter approach is useful, for example, during parallel finite element matrix assembly on an adaptively refined mesh. In this case, the user obtains iterators which traverse the set of active elements (described in more detail in section 5.4.3) which are owned by the local processor.

The mesh class is designed to be extensible. Encapsulating the stored elements and nodes by providing access only through custom iterators admits the possibility of providing different implementations for specific instances. The `Mesh` implementation assumes a fully unstructured, hybrid element mesh. However, algorithmic and storage-based optimizations for Cartesian grids, block-structured grids, and grids with only a single type of element could be added without changing the current interface.

5.2 Degrees of Freedom

The first finite elements implemented in `libMesh` were the standard Lagrange elements with nodal value degrees of freedom. The library has since been extended to a wider variety of finite element types (see section 5.6). Shape functions on more exotic finite elements can correspond to nodal Hessian components, mid-edge normal fluxes, or orthogonal hierarchic polynomials. For these finite element types, it no longer makes sense to associate each shape function with a single geometric point.

The `DofObject` class handles these different types of degrees of freedom generically. Examples of `DofObjects` are element interiors, faces, edges, and vertices. An element interior has associated degrees of freedom for those shape functions whose support is contained within the element. Face degrees of freedom correspond to shape functions contained within the two elements sharing a face, edge degrees of freedom correspond to shape functions for all elements sharing an edge, and vertex degrees of freedom correspond to shape functions supported on all elements sharing a single vertex.

The domain decomposition approach described earlier assigns disjoint groups of elements to individual processors. This allows the element-based degrees of freedom to be assigned uniquely to the processor which owns the element, but requires some shared distribution of vertex, edge, and face degrees of freedom. Figure 4 illustrates the approach which is used in the library. In this approach, any degrees of freedom on the border between subdomains are owned by the processor of lowest global index. This is evident from the figure, where the nodes on the shared interface have been assigned to processor 0.

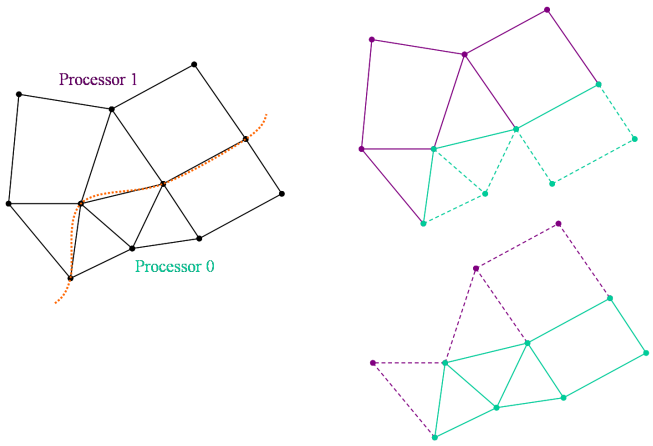

Fig. 4 Element partitioning & degree of freedom distribution. Disjoint element sets are divided between processors, while boundary nodes are assigned to the processor with lower ID.

This approach for assigning degrees of freedom to processors also fits well with the sparse matrix partitioning scheme employed in PETSc, where complete rows of the sparse matrix are assigned to individual processors [47]. This is the natural matrix decomposition that results from the degree of freedom distribution used in the library.

5.3 Nodes

Each object of the `Node` class stores its (x, y, z) location in space, as well as additional state information including a unique global identification number (ID) and degree of freedom indices. The mesh data structure contains a complete list of all nodes. Nodes may be accessed directly by the user via iterators, or indirectly through elements which are connected to the nodes. Trivial operations such as scaling, translating, or rotating a mesh are performed directly on the nodes.

During the refinement process new nodes may be added to the mesh. When two adjacent elements are refined, common nodes will exist on the inter-element interface. This situation must be properly resolved to achieve a valid discretization (i.e. with no duplicate nodes). A new node is created as a linear combination of existing nodes, and a hash key is constructed for each new node based on the weights and global IDs of its parent nodes. If this key already exists in the map of new node keys, the new node is a duplicate and is therefore rejected. This procedure efficiently resolves nodal connectivity for refined elements.

Similarly, coarsening the mesh can create “orphan nodes,” or nodes that are not connected to any elements. After an

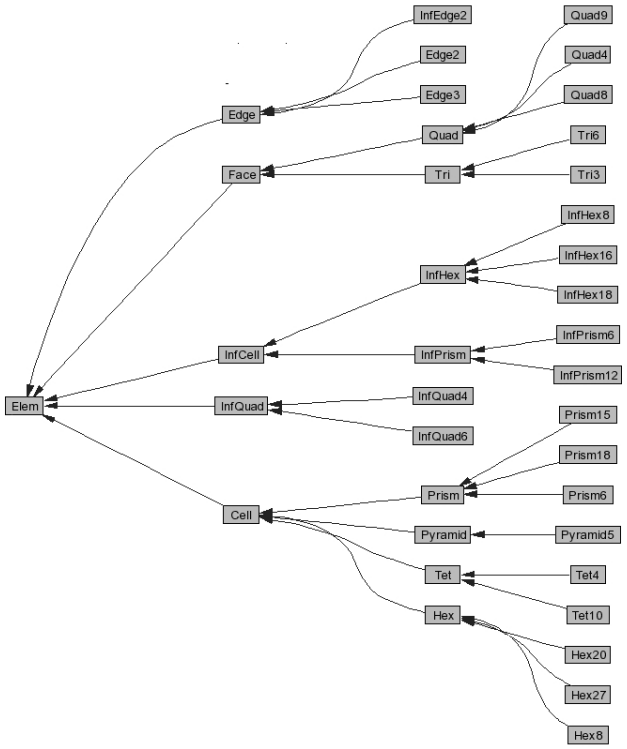

Fig. 5 The `Elem` class hierarchy

AMR/C step the library simply counts the number of elements connected to each node and removes those nodes which are not connected to any elements.

5.4 Elements

`libMesh` defines the abstract base class `Elem` which defines the interface for a geometric element. Concrete subclasses of `Elem`, such as `Quad4` and `Tet10`, are specialized via virtual function calls to return e.g. the correct number of nodes and sides when `n_nodes()` and `n_sides()` are called on an `Elem` pointer. The complete list of geometric element types provided in `libMesh` is shown in Figure 5. Note that an `Edge` *is* an `Elem` (in the polymorphic sense) in 1D, and similarly for `Face` in 2D and `Cell` in 3D. Implementations of all the standard geometric element types used in finite element analysis including quadrilaterals, triangles, hexahedra, tetrahedra, prisms, and pyramids, as well as a collection of infinite elements, are provided in `libMesh`.

5.4.1 Nodal Connectivity Elements contain state information similar to nodes. Elements store a unique ID, their processor ID, and degree of freedom information. Additionally, the element connectivity is stored as pointers to nodes. This is a slight departure from the classic finite element data structure, in which the element connectivity is defined in terms of the nodal indices [4]. On 32-bit machines pointers and integers are both 4 bytes, so this choice does not impose additional storage. On 64-bit machines, however, pointers are 8 bytes, which essentially doubles the amount of memory required to store element connectivity.

This approach for storing the element connectivity was chosen so that elements could have increased functionality in the absence of a corresponding `Mesh` object. A traditional connectivity scheme would require the mesh to access the nodal locations of a given element. This is important, for example, when computing the map from a physical to reference element or determining if a point lies inside an element. By storing pointers to the nodes, the element can determine its geometric connectivity directly. This simplifies many functions in the code by requiring the user to pass only an element instead of both an element and the nodal locations. Additionally, this approach reduces the amount of indirect memory addressing required for an element to obtain nodal information.

5.4.2 Face Neighbors Elements also store pointers to their face neighbors. Two elements are said to be face neighbors if they share a "side," where a "side" is a `Node` in 1D, an `Edge` in 2D, and a `Face` in 3D. If an element side is on the physical boundary of the domain there will be no neighbor. Locating the elements coincident with the boundary is equivalent to finding all the elements which have at least one side with no neighbor. This is useful when applying boundary conditions.

After reading a mesh from disk, or performing mesh refinement, it is necessary to construct the face neighbor information efficiently. The library handles this by looping over all the elements and then over the sides of the elements. If a neighboring element has not been located already the side of the element is constructed and a hash key is computed based on the global indices of its nodes. A map is then queried to find any elements with sides matching this key, and they are checked for a possible match. The loop through the N elements is $O(N)$, while for a map of size M the lookup is $O(\log M)$, so the resulting algorithm has $O(N \log M)$ complexity. With $M \leq N$, this yields a potentially $O(N \log N)$ algorithm. Alternate approaches are possible for which $M \ll N$ which could improve performance for very large meshes. For example, ordering the elements with a space-filling curve before performing the neighbor search will ensure adjacent elements are quickly located, reducing the overall size of the map.

Since constructing the side of an element is a common task, a special proxy class called `Side` has been developed for this purpose. This class essentially defines the side of an element as a new element living in a lower spatial dimension and provides the connectivity through a mapping from the original element. This approach allows the side of an element to be constructed rapidly, as the allocation and population of a new connectivity array is not required.

5.4.3 Element Refinement Hierarchy Elements are refined upon user request via the "natural refinement" scheme. In this approach d-dimensional elements are generally refined into 2^d subelements of the same type. (Pyramid refinement is

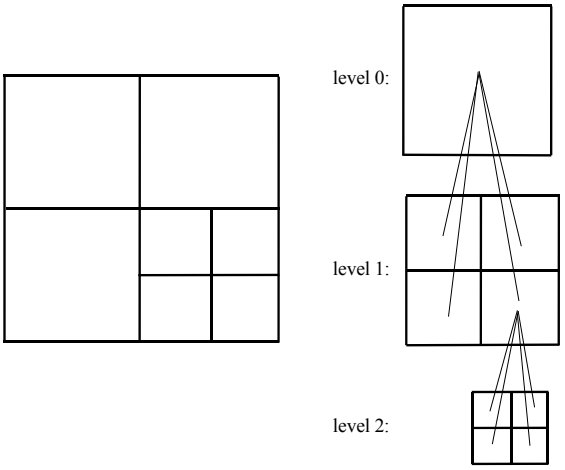

Fig. 6 Element refinement hierarchy and resulting quadtree for a 2D quadrilateral mesh.

an exception to this rule: refining a pyramid results in a collection of pyramids and tetrahedral elements.) Hanging nodes are allowed at element interfaces and hanging degrees of freedom are constrained algebraically. This approach was chosen because it is applicable for general hybrid meshes with arbitrary types of elements, and in general results in refined elements of the same type. This latter point ensures that refining an all-quad mesh in 2D produces an all-quad mesh, for example.

This refinement approach naturally yields a tree data structure, and Figure 6 shows the quad tree data structure which results from refining a single quadrilateral element. Each element has a pointer to its "parent," and an array of pointers to its "children." The initial, level-0 elements are unique in that they have no parent. Similarly, the active elements which are used in finite element computations have no children. The level of a given element is determined recursively from its parent. The user is allowed to access any subset of the elements via iterators as discussed previously. The active elements are commonly used in matrix assembly, but intermediate levels could also be used in a multigrid cycle, for example.

The element hierarchy is additionally used to locate hanging nodes in the mesh which must be constrained. As mentioned previously, elements store pointers to neighboring elements which share sides. These neighboring elements are necessarily at the same level of refinement. If an active element's neighbor is a refined element, then any degrees of freedom located on the common side must be constrained.

The refinement hierarchy also naturally supports element coarsening. In the case that all of the children of an element are flagged for coarsening, the parent element simply deletes its children and becomes active again. In Figure 6, this would correspond to all the level-2 elements being deleted. The resulting mesh would contain just the active level-1 elements and their parent. A consequence of this approach to element coarsening is that the mesh cannot be coarsened below the initial, level-0 mesh. In many cases it is desirable to use the coarsest level-0 mesh possible and allow the refinement process to add elements only where they are needed.

5.5 Systems

The abstract `System` class in `libMesh` corresponds to a PDE system of one or more equations that is to be solved on a given mesh. `libMesh` provides several concrete system implementations including explicit, implicit, steady, transient, linear, and nonlinear systems. A system stores the solution values for the degrees of freedom in a simulation, which may be either real- or complex-valued. Additionally, a system may contain additional information such as a sparse matrix, which is required for implicit solution strategies. In the current implementation a system is uniquely tied to a given mesh, so a simulation that uses multiple meshes must also solve multiple systems.

The `System` class provides a generic, customizable interface which allows the user to specify the physics-dependent parts of an application. For example, in the case of an implicit system users can provide a function for matrix assembly or can derive their own class and overload the matrix assembly operator. Similarly, for transient systems the user may either provide an initialization function or overload the initialization operator provided in the library.

Multiple systems may be tied to a given mesh to allow for loose coupling of different physics. This feature has been applied in the case of Rayleigh Bénard Marangoni flows to decouple the incompressible fluid flow and heat transfer equations. In this example two implicit systems are solved in an iterative fashion. Similarly, incompressible flows using pressure projection operator-splitting techniques have been solved using a combination of explicit and implicit systems.

The library makes extensive use of C++ templates to allow complicated systems to be constructed from simpler subsystems. For example, transient nonlinear systems are supported by combining a transient outer loop with a nonlinear inner loop. Templates are useful in this setting because they allow simple components to be combined into a complex algorithm. This enhances code reuse and minimizes debugging efforts.

5.6 Finite Element Spaces

The library provides a number of finite element "families" that may be used in a simulation. The classic first and second order Lagrange finite elements are supported, as well as C^0 hierarchic elements of arbitrary polynomial order. Mapping between physical and computational space is performed with the Lagrange basis functions that are natural for a given element. For example, mapping of a three-node triangle is performed with the linear Lagrange basis functions, while a 27-node hexahedral element is mapped with a tri-quadratic Lagrange basis. For many mesh geometries, quadratic Lagrange elements are only mapped linearly from computational space.

Provisions are made in the library to detect this and use the minimal polynomial degree required for an accurate map.

Discontinuous finite element spaces are also supported. For these approximation spaces the degrees of freedom are wholly owned by the elements. The library offers monomial finite element bases for these spaces. One approach is to use the monomial basis defined in terms of the reference element (ξ, η, ζ) coordinates for each element in the domain. Another option is to use the physical (x, y, z) coordinates inside the element as the monomial basis. The former approach is efficient when the discontinuous spaces will be used primarily for integration inside the element (such as the LBB-stable Q_2P_{-1} quadrilateral element for incompressible flows [48]), while the latter approach is attractive for the many element boundary computations which arise in the discontinuous Galerkin family of finite element methods and in finite volume discretizations.

Support for C^1 continuous elements is provided in the library. Users can generate Clough-Tocher [49] and reduced Clough-Tocher [50] triangular macroelements on arbitrary 2D meshes, as well as tensor products of cubic or higher Hermite polynomials on rectilinear meshes in up to 3 dimensions. Either element choice gives a function space with continuous values and first derivatives, suitable for the solution of fourth-order problems posed on $W^{1,p}$ spaces. In all cases, h adaptivity is not precluded and the library can constrain hanging degrees of freedom to produce C^1-conforming functions on hanging node meshes. The Hermite-based elements support C^1 function spaces on p and hp adapted meshes as well, and future work will add this capability to more general C^1 elements.

`libMesh` also provides Astley-Leis infinite elements for the analysis of unbounded domains, such as sound radiation of vibrating structures [51]. The infinite elements may be generated on top of the outer surface of a previously generated finite element mesh. The transformation from the physical space is performed using a $1/r$-like mapping, where r is the radial (infinite) direction, combined with conventional finite element shape functions on the base of an infinite element. The user may chose between different radial polynomial bases [52], where shape approximations up to eighteenth order are implemented. The element hierarchy shown in Figure 5 was easily extended to account for these classes of elements and associated refinement rules, so adding support for these special classes of elements was fairly straightforward within the `libMesh` design.

The user specifies the finite element family and the initial approximation order (before any p refinement) to be used for each variable in a system. The abstract `FEBase` class provides the generic interface for all finite element families, and specific cases are instantiated with template specialization. The `FEBase` class provides essential data for matrix assembly routines such as shape function values and gradients, the element Jacobian, and the location of the quadrature points in physical space. These calculations were implemented in the library to simplify users' physics code, but as an additional benefit this modularity has allowed many `libMesh` upgrades, from C^1 function spaces to p adaptivity support, to be accessible to users without requiring changes to their physics code.

Templates are used extensively in the finite element hierarchy to reduce the potential performance overhead of virtual function calls. There are other tradeoffs to consider when using templates, however, such as the size of the resulting object files and the difficulty of programming new finite elements without all of the benefits of polymorphism. Detailed profiling studies on the benefits of refactoring the finite element hierarchy are the subject of future work.

6 Finite Element Independent Adaptivity

A primary goal of `libMesh` is extensibility: it should be easy for experienced users to add new finite element types to the system with minimal effort. To make this possible, `libMesh` includes element-independent implementations for hanging node constraints, solution restrictions to coarsened meshes, and solution projections to refined meshes. When adding a new finite element to the library, developers can first use these default implementations, only replacing them with element-specific implementations if necessary for efficiency.

6.1 Hanging Node Constraints

When using the hierarchical mesh refinement capabilities provided by `libMesh`, the resulting meshes are non-conforming, with "hanging nodes" on sides where coarse elements and more refined elements meet. On these sides, the spaces of function values and fluxes on the coarse element are strict subspaces of the values and fluxes which are possible on the refined neighbors. Ensuring C^r continuity between these spaces requires constraining some or all of the refined element degrees of freedom.

Degrees of freedom on the side of a fine element must be expressed in terms of degrees of freedom on the overlapping side of a neighboring coarse element. The goal is to ensure that all function values and derivatives up to the required continuity level are equal. We impose this constraint in an element-independent way by forming and solving L_2 projection problems for the solution values and for all continuous solution derivatives across a side.

The construction and numerical inversion of these small matrices is less computationally efficient than specialized constraint matrix construction based on specific element degree of freedom equations, but a single projection-based constraint code can be applied to any new finite element object whose shape functions have been programmed. This offers greater support for implementors of new finite element types.

6.2 Refinement and Coarsening

Adaptive mesh coarsening requires the restriction of solution data onto a coarse parent element based on the approximate

solution on its refined children, and adaptive mesh refinement requires the projection of solution data onto refined child elements from their original coarse parent. The restriction and projection operator should be as accurate as possible, but just as importantly the operator should be computationally efficient, uniquely defined, parallelizable, and independent of finite element type. We again use Hilbert space projection operators to maintain that independence. Using an element-wise L_2 or H^1 projection is efficient, runs in parallel without inter-processor communication (given the data dependencies discussed in Section 7.1), and gives an exact solution in the case of refinement using nested finite element spaces. For coarsening or for refinement in non-nested spaces, however, an element-wise Hilbert projection would not be uniquely defined, since the projections from neighboring cells could produce different function values along their shared side.

A more complicated but similarly efficient algorithm restores uniqueness by acting on these shared degrees of freedom first, as follows: We start by interpolating degrees of freedom on coarse element vertices. Holding these vertex values fixed, we do projections along each coarse element edge. Because these projections involve only data from the original refined elements on that edge and not data from element interiors, they are uniquely defined. In 3D, element faces are then projected while holding vertex and edge data fixed. Finally, element interior degrees of freedom are projected while holding element boundary data fixed. Although the preceding series of projections is more complicated than a single per-element projection, the number of degrees of freedom to be solved for at each stage is much smaller, and so the dense local matrix inversions required are faster. These projections each only require local element data and so are as easy to parallelize as whole-element projections, but because the node, edge, and face projections give uniquely defined results for degrees of freedom shared between elements, when `libMesh` calculates them in parallel it will still arrive at consistent results.

7 Parallel Issues

Parallelism in `libMesh` is exploited at the matrix assembly and linear algebra levels. On distributed memory machines, such as PC clusters, a complete copy of the mesh is maintained independently on each processor. This design decision limits practical 3D applications to on the order of 128 processors because of the overhead associated with storing the global mesh. Nevertheless a remarkable number of 3D applications have been successfully solved using this implementation, and keeping a copy of the mesh on each processor mitigates some of the load balancing issues that fully-parallel mesh data structures must contend with. The recent development of hybrid distributed/shared memory architectures, such as PC clusters with multi-core CPUs, suggests that corresponding parallel codes should include combined message passing and multithreading models.

A major goal of the library is to shield end-users from the complexity of parallel programming, allowing them instead to focus on the physics they are modeling. The vision is for users to develop and debug applications on serial machines and then move seamlessly to parallel architectures for large-scale simulations. To achieve this goal the library hides parallel communication from the user, so basic MPI calls are not required in most applications.

A case in point is the simple act of reading a mesh from disk. The user simply instantiates a mesh object and calls its `read()` member function. This is a trivial operation from the user's point of view, consisting of only two lines of code. These two lines of code are then executed on every processor in a parallel simulation, causing processor 0 to actually read the file from disk and send (via `MPI_Bcast`) the data to the remaining processors. This level of abstraction is common in many numerical libraries (e.g. PETSc) which use MPI.

7.1 Data Dependencies

The degree of freedom distribution discussed in section 5.2 allows for shared degrees of freedom on processor boundaries. This allows local elements to both depend on and contribute to remote degrees of freedom. Hence, we require some synchronization process to obtain remote data.

For a classic finite element discretization, computations on a given element are dependent solely on the element's own degrees of freedom. Synchronizing only the shared degrees of freedom is sufficient in this case. However, certain error indicators and discontinuous Galerkin schemes compute the interface flux jump, which also depends on all the degrees of freedom in a neighboring element. For this reason `libMesh` synchronizes not only shared degrees of freedom but all the degrees of freedom corresponding to the face neighbors of the local elements. This corresponds to all the degrees of freedom for the "ghost" elements depicted in Figure 4.

Synchronization is performed in the library after the completion of a solve step. For example, the completion of a linear solve will result in updated degrees of freedom on each processor, and a communication step is required so that updated values for remote degrees of freedom are obtained. The library performs this step at the end of each solve without any user intervention.

7.2 Matrix Assembly

The domain decomposition approach used in the library naturally lends itself to parallel matrix assembly. The matrix assembly code provided by the user operates on the active elements local to each processor. The standard approach of assembling element matrices into the global matrix for an implicit solution strategy is used. In this approach the data needed to assemble the local element matrices is collected before the assembly procedure, and the actual matrix assembly can be performed in parallel.

The degree of freedom distribution used in the library permits local element matrices to contribute to remote degrees of freedom for elements on inter-processor boundaries. Hence,

communication may be required in forming the global matrix. In PETSc, sparse matrix objects accumulate entries that must be communicated during the matrix assembly phase and cache them, which prevents costly inter-processor communication for each element in the assembly loop. After each element matrix is inserted on a given processor, communication is required to correctly sum the entries for these shared degrees of freedom. The matrix assembly phase can be summarized by the following steps:

1. Synchronize data with remote processors. This is required so that any remote data needed in the element matrix assembly is available on the local processor.
2. Perform a loop over the active elements on the local processor. Compute the element matrix and distribute it into the global matrix.
3. Communicate local element contributions to degrees of freedom owned by remote processors.

The first and third steps are performed automatically by the library, while the second step requires user-supplied code for forming the element matrices, or for residual evaluation in the case of Jacobian-free Newton-Krylov methods.

8 Applications

One aspect of the finite element method which `libMesh` certainly reflects is its wide ranging applicability. This, combined with the open source development method, has fostered application work in a geographically and scientifically diverse number of areas. The original CFDLab developers have used `libMesh` for incompressible Navier-Stokes applications including thermocapillary natural convection (see Figs. 7 and 8) and shear-thinning flows. Compressible Euler (Fig. 9) and Navier-Stokes applications, including aerothermodynamics research for orbiter reentry at NASA, have also been conducted with `libMesh` using both SUPG and discontinuous Galerkin formulations. Different models for flow in porous media, including the Elder problem and the double-diffusive natural convection problem shown in Fig. 10, have been studied, as have biological simulations of *e. Coli* proliferation and tumor angiogenesis models (Fig. 11). Other applications being simulated using `libMesh` include compressible boundary layer calculations [53], plate bending, linear advection diffusion reaction, Stokes flow, and Burgers' equation.

Application areas for the wider `libMesh` community include electrostatics in thin films of silicon and composite materials, linear elasticity with Cauchy-Born constitutive models, Stokes flow with free capillary boundaries, optical imaging, Helmholtz and wave equations for interior and exterior domains, eigenvalue/modal analysis, 3D geoelectric solvers with infinite elements for potential field continuation, magnetic resonance simulation, nonlinear heat conduction, cavity radiation, thermoelastic problems in solid mechanics, calcium dynamics in cardiac cells, and Lagrangian particle tracking. For additional references in which `libMesh` was used as part of the solution methodology, see [8, 51, 55–61].

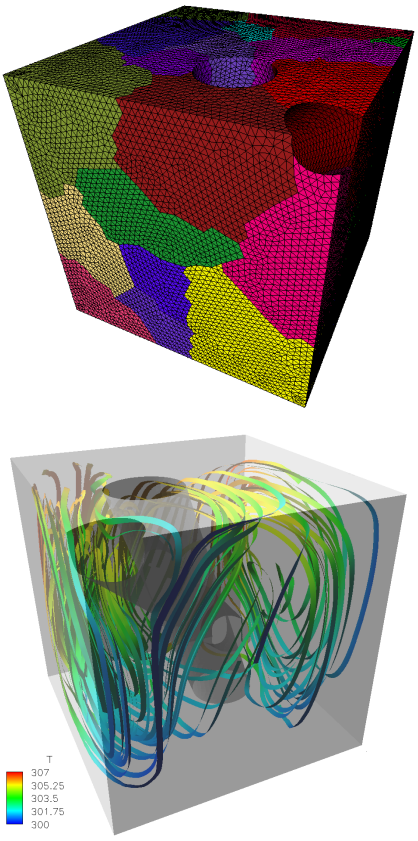

Fig. 7 Buoyancy driven flow in a complex geometry, solved in parallel on a workstation cluster. The upper figure shows the METIS partitioning of a tetrahedral mesh interior to a cube domain and exterior to two cylindrical "pipes". The lower figure depicts stream ribbons colored by temperature. The fluid is naturally convected away from the hot wall of the domain and forms a complex circulation field around the pipe geometry.

It is important to note that solution algorithms are necessarily highly problem-dependent. This is underscored by contrasting the solution algorithms used for compressible flows (figure 9) and incompressible flows (figures 7 and 8). The nonlinear problem arising in implicit algorithms for compressible flows is notoriously sensitive to the initial guess, hence time stepping to steady-state is a common technique for solving these problems. At a given time step the resulting nonlinear problem is only approximately solved. For this application, the run-time is essentially split between matrix assembly and executing linear solves. By contrast, steady incompressible flows result in a nonlinear system which is considerably less sensitive to initial guess. These applications may be solved either steady or via time marching with a small number of time steps. In this case, the nonlinear problem is solved to much higher accuracy. A typical incompressible flow application may spend 15% of run-time in matrix assembly with the remaining 85% spent in solving the linear system to a high accuracy.

`libMesh` provides a wide range of building blocks for steady or transient, linear or nonlinear, implicit or explicit,

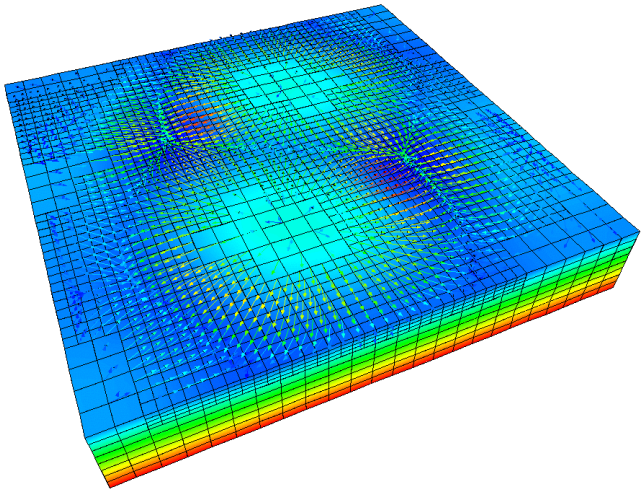

Fig. 8 Thermocapillary surface-tension (Rayleigh-Bénard-Marangoni) flow application with adaptivity, solved in parallel on a workstation cluster. Temperature contours are shown, with warmer fluid rising from the bottom of the domain due to buoyancy and then spreading when it reaches the surface due to thermocapillary effects. Also shown are surface velocity vectors and localized refinement driven by velocity gradients at the developing convection cell boundaries.

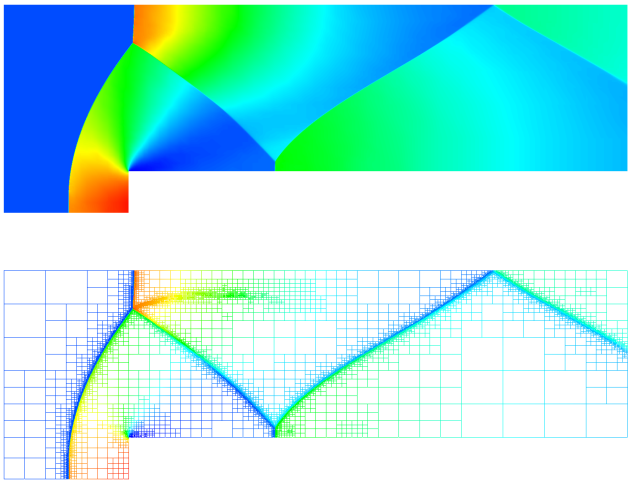

Fig. 9 Pressure field for Mach 3 inviscid flow over a forward facing step. In this case, the adaptivity is driven by inter-element jumps in velocity and tracks normal and oblique shock waves in the flow. The contact surface emanating from the Mach stem near top of the domain, a constant pressure structure separating regions of supersonic and subsonic flow, is naturally tracked by the indicator as well.

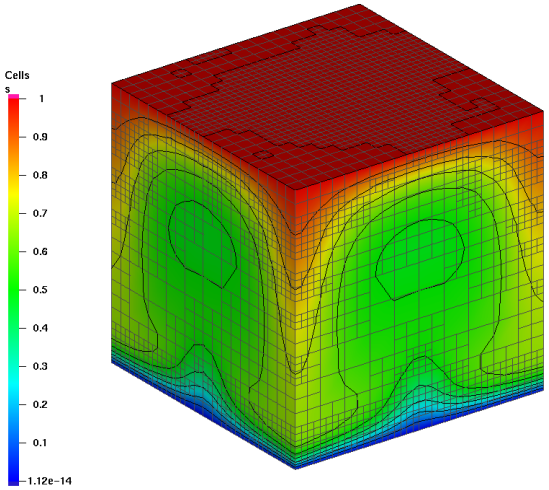

Fig. 10 Solute contours in a 3D adaptive simulation of double-diffusive convection in a porous medium. A plume of warm, low concentration fluid is convected upward, and a solute boundary layer develops near the bottom of the domain. The adaptivity is driven by a physics-independent indicator as discussed in Section 2, which in this case is related to inter-element jumps in the solutal flux.

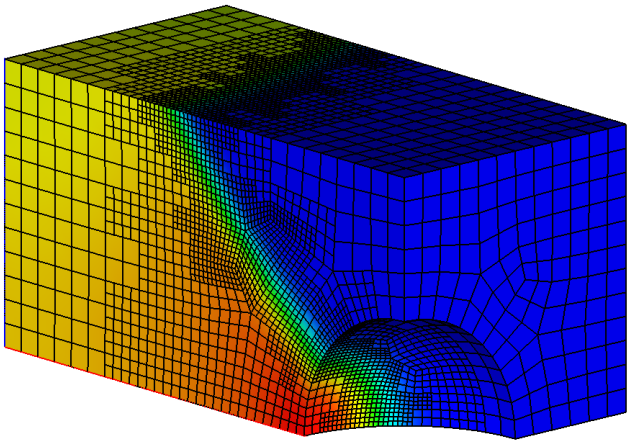

Fig. 11 Adaptive 3D solution to the tumor angiogenesis problem conducted on 64 processors. This application models an extension of the 2D model considered by Valenciano and Chaplain [54]. The tumor, represented by the cut-out spherical region, secretes a chemical which attracts endothelial cells (represented by the contours) and eventually leads to the birth of new blood vessels which feed the tumor. In this simulation, AMR using a physics-independent error indicator tracks the advancing front of endothelial cells.

static or dynamic mesh algorithms (and combinations thereof). As mentioned previously, many algorithmic details such as linear solver tolerances, refinement criteria, mesh partitioning quality, etc. interplay in these advanced applications. It is not appropriate for a physics-independent library to make these choices, and thus in `libMesh` they are controlled by the user. Numerical experiments are key for finding the optimal solution algorithm for a given application.

9 Concluding Remarks and Future Plans

As illustrated in the applications sample, `libMesh` provides a powerful capability for efficient and accurate adaptive finite

element solutions of diverse applications in a serial or parallel environment. It requires a nominal initial effort by the applications analyst to encode in C++ a Jacobian and residual description, and some understanding of the application to select better tolerances than the default values may provide. The library permits AMR/C simulations on different architectures including Linux clusters and can handle hybrid meshes using a 2-level AMR/C scheme with hanging nodes. It has been tested by the CFDLab members for Galerkin, Petrov-Galerkin, and Discontinuous Galerkin schemes. Standard C^0 and C^1 finite elements as well as infinite elements are supported.

Some of the issues that we are addressing include recovery and other physics-independent error indicators. Future studies may involve closer linkage to specific application codes, the use of more sophisticated dual indicators, and the development of error indicators suited to fully automatic hp adaptivity. Augmentation by mesh smoothing and redistribution is also an interesting area, particularly from the standpoint of adaptivity, since smoothing can be conducted at the coarsest mesh level. Combined adaptive refinement-redistribution-smoothing techniques will likely require conforming and/or anisotropic refinement strategies, both of which are areas for future library improvement. The algorithm for simultaneous refinement and coarsening is also being improved. This will involve studies regarding the selection of tolerances for refinement and solution steps, and impact the frequency of dynamic repartitioning.

Finally, a fully parallelized implementation of the basic unstructured mesh data structure is being explored. The current implementation duplicates the global mesh on each processor and is clearly a limitation for scalability and maximum problem size. Also, mesh class specializations for Cartesian and block-structured grids are also being considered. The mesh data structures have been designed to allow for specific implementations to handle these special cases.

10 Acknowledgments

The student authors of `libMesh` have been partially supported by a Department of Energy Computational Science Graduate Fellowship, Institute for Computational and Engineering Sciences (ICES) fellowships, NASA Graduate Student Research Grant NGT5-139, and DARPA Grant No. HR-0011-06-1-0005.

David Knezevic performed the tumor angiogenesis simulation and implemented support for 1D problems in the library. Varis Carey provided a patch recovery error indicator implementation. Infinite elements, support for complex-valued systems, and eigenvalue problems were provided by Daniel Dreyer and Steffen Petersen from Technische Universität Hamburg-Harburg. Additionally, we are grateful to the `deal.II` project for inspiring `libMesh`, and Wolfgang Bangerth in particular for many useful discussions.

References

1. Gropp W, Lusk E, Doss N, Skjellum A (1996) MPICH: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing 22(6):789–828
2. Carey GF (1997) Computational Grids: Generation, Adaptation, and Solution Strategies. Taylor & Francis
3. Barth W, Kirk B (2003), PC Cluster Construction: Lessons Learned and Friendly Advice. Shortcourse on Cluster Computing and Adaptive Finite Element Methods, The University of Western Australia, Perth, Australia
4. Becker EB, Carey GF, Oden JT (1981) Finite Elements – An Introduction, vol. 1. Prentice Hall
5. Flaherty JE, Paslow PJ, Shephard MS, Vasilakis JD, editors (1989) Adaptive Methods for Partial Differential Equations. SIAM
6. Babuska I, Zienkiewicz OC, Gago J, Oliviera ERA (1986) Accuracy Estimates and Adaptive Refinements in Finite Element Computations. Wiley
7. Babuska I, Rheinboldt WC (1982) Computational Error Estimates and Adaptive Processes for Some Nonlinear Structural Problems. Computer Methods in Applied Mechanics and Engineering 34:895–937
8. Carey GF, Anderson M, Carnes B, Kirk B (2004) Some aspects of adaptive grid technology related to boundary and interior layers. J Comput Appl Math 166(1):55–86, ISSN 0377-0427, doi:http://dx.doi.org/10.1016/j.cam.2003.09.036
9. Barth W, Carey GF, Kirk B, McLay R (2000) Parallel Distributed Solution Of Viscous Flow With Heat Transfer On Workstation Clusters. In High Performance Computing '00 Proceedings, Washington, D.C.
10. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley
11. Carey GF, McLay R (1995) Multi-Level Preconditioners for the Parallel PCG Package, Copper Mtn. Conf. on Multigrid Methods. In Proceedings of the Copper Mountain Conference on Multigrid Methods, Copper Mountain, CO
12. Carey GF (1976) A Mesh Refinement Scheme for Finite Element Computations. Computer Methods in Applied Mechanics and Engineering 7:93–105
13. Carey GF (1976) An Analysis of Finite Element Equations and Mesh Subdivision. Computer Methods in Applied Mechanics and Engineering 9:165–179
14. Carey GF, Finlayson BA (1975) Orthogonal Collocation on Finite Elements. J Chemical Engineering Science 30:587–596
15. Bank RE (1998) PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, Users' Guide 8.0, Software, Environments and Tools, Vol. 5. SIAM, Philadelphia
16. Plaza A, Carey GF (2000) Local refinement of simplicial grids based on the skeleton. Applied Numerical Mathe-

matics 32:195–218
17. Plaza A, Padrón MA, Carey GF (2000) A 3D refinement/derefinement algorithm for solving evolution problems. Applied Numerical Mathematics 32:401–418
18. Carey GF, Kabaila A, Utku M (1982) On Penalty Methods for Interelement Constraints. Computer Methods in Applied Mechanics and Engineering 30:151–171
19. Bangerth W, Rannacher R (2003) Adaptive Finite Element Methods for Differential Equations. Birkhäuser Verlag, Basel, Switzerland
20. Eriksson K, Estep D, Hansbo P, Johnson C (1996) Adaptive Finite Elements. Springer-Verlag, Berlin
21. Estep D, Larson M, Williams R (2000) Estimating the error of numerical solutions of systems of nonlinear reaction–diffusion equations. Memoirs of the American Mathematical Society 696:1–109
22. Estep D, Holst M, Mikulencak D (2002) Accounting for stability: a posteriori estimates based on residuals and variational analysis. Communications in Numerical Methods in Engineering 18:15–30
23. Estep D, Holst M, Larson M (2005) Generalized Green's functions and the effective domain of influence. SIAM Journal on Scientific Computing 26:1314–1339
24. Kelly DW, Gago JP, Zienkiewicz OC, Babuska I (1983) A posteriori error analysis and adaptive processes in the finite element method: Part I Error analysis. Int J Num Meth Engng 19:1593–1619
25. Zienkiewicz OC, Zhu JZ (1987) A simple error estimator and adaptive procedure for practical engineering analysis. International Journal for Numerical Methods in Engineering 24:337–357
26. Wahlbin LB (1995) Superconvergence in Galerkin Finite Element Methods. Springer
27. Carey V (2005) A Posteriori Error Estimation for the Finite Element Method via Local Averaging. Ph.d. thesis, CAM dept., Cornell University
28. Stewart JR, Edwards HC (2004) A framework approach for developing parallel adaptive multiphysics applications. Finite Elem Anal Des 40(12):1599–1617, ISSN 0168-874X, doi:http://dx.doi.org/10.1016/j.finel.2003.10.006
29. Bangerth W (2000) Using Modern Features of C++ for Adaptive Finite Element Methods: Dimension-Independent Programming in deal.II. In M Deville, R Owens, editors, Proceedings of the 16th IMACS World Congress 2000, Lausanne, Switzerland, 2000, Document Sessions/118-1
30. Budge K, Peery J (1996), Experiences Developing ALEGRA: A C++ Coupled Physics Framework. In M.E Henderson, C. R. Anderson, and S. L. Lyons, editors, Object Oriented Methods for Interoperable Scientific and Engineering Computing
31. Barton JJ, Nackman LR (1994) Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples. Addison Wesley
32. Bastian P, et al (1997) UG – A flexible software toolbox for solving partial differential equations. Computing and Visualization in Science 1:27–40
33. Devloo PRB, Longhin GC (2002) Object oriented design philosophy for scientific computing. Mathematical Modelling and Numerical Analysis 36:793–807
34. Demkowicz L, Rachowicz W, Devloo PRB (2002) A fully automatic hp-adaptivity. J Scientific Computing 17:127–155
35. Free Software Foundation (1999), GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
36. Imamura M (2002) Using Doxygen: A quick guide to getting started and using the Doxygen inline documentation system for documenting source code. Tech. rep., Linux Users Group at Georgia Tech
37. Meyers RJ, Tautges TJ, Tuchinsky PM (1998) The Hex-Tet Hex-Dominant Meshing Algorithm as Implemented in CUBIT. In Proceedings of the 7th International Meshing Roundtable, 151–158
38. Shewchuk JR (1996) Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In MC Lin, D Manocha, editors, Applied Computational Geometry: Towards Geometric Engineering, vol. 1148 of *Lecture Notes in Computer Science*, 203–222, Springer-Verlag, from the First ACM Workshop on Applied Computational Geometry
39. Si H (2005) TetGen - A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator. Weierstrass Institute for Applied Analysis and Stochastics, Berlin
40. Karypis G, Kumar V (1995) METIS unstructured graph partitioning and sparse matrix order. Tech. rep., University of Minnesota, Department of Computer Science
41. Karypis G, Kumar V (1998) A parallel algorithm for multilevel graph partitioning and sparse matrix reordering. Parallel and Distributed Computing 48:71–95
42. Devine K, Boman E, Heaphy R, Hendrickson B, Vaughan C (2002) Zoltan Data Management Services for Parallel Dynamic Applications. Computing in Science and Engineering 4(2):90–97
43. Skalický T (1996) LASPack Reference Manual. Dresden University of Technology
44. Balay S, Eijkhout V, Gropp WD, McInnes LC, Smith BF (1997) Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E Arge, AM Bruaset, HP Langtangen, editors, Modern Software Tools in Scientific Computing, 163–202, Birkhäuser Press
45. Hernandez V, Roman JE, Vidal V (2005) SLEPc: A Scalable and Flexible Toolkit for the Solution of Eigenvalue Problems. ACM Trans Math Softw 31(3):351–362
46. Iqbal S, Carey GF (2005) Performance analysis of dynamic load balancing algorithms with variable number of processors. Journal of Parallel and Distributed Computing 65(8):934–948
47. Balay S, Buschelman K, Eijkhout V, Gropp W, Kaushik D, Knepley M, McInnes LC, Smith B, Zhang H (2005) PETSc Users Manual. Argonne National Laboratory,

Mathematics and Computer Science Division, 2.3.0 edn.
48. Gresho PM, Sani RL (1998) Incompressible Flow and the Finite Element Method. John Wiley and Sons
49. Clough R, Tocher J (1965) Finite element stiffness matrices for analysis of plates in blending. In Proceedings of Conference on Matrix Methods in Structural Analysis
50. Ciarlet PJ (1978) The Finite Element Method for Elliptic Problems. North-Holland, Amsterdam
51. Dreyer D, Petersen S, von Estorff O (2006) Effectiveness and robustness of improved infinite elements for exterior acoustics. Computer Methods in Applied Mechanics and Engineering 195(29-32):3591–3607
52. Dreyer D, von Estorff O (2003) Improved conditioning of infinite elements for exterior acoustics. International Journal for Numerical Methods in Engineering 58(6):933–953
53. Marichalar JJ, Rochelle WC, Kirk BS, Campbell CH (2006) BLIMPK/Streamline Surface Catalytic Heating Predictions on the Space Shuttle Orbiter, 44th AIAA Aerospace Sciences Meeting and Exhibit, AIAA Paper 2006-180
54. Valenciano J, Chaplain MAJ (2004) An explicit subparametric spectral element method of lines applied to a tumor angiogenesis system of partial differential equations. Mathematical Models and Methods in Applied Sciences 14:165–187
55. Carey G, Barth W, Woods JA, Kirk BS, Anderson ML, Chow S, Bangerth W (2004) Modelling error and constitutive relations in simulation of flow and transport. International Journal for Numerical Methods in Fluids 46:1211–1236
56. Peterson JW, Carey GF, Knezevic DJ, Murray BT (accepted Mar. 2006) Adaptive finite element methodology for tumor angiogenesis modeling. Int J Numer Meth Eng
57. Simedrea P, Antiga L, Steinman DA (2006) Towards a New Framework for Simulating Magnetic Resonance Imaging. In First Canadian Student Conference on Biomedical Computing, http://cscbc2006.cs.queensu.ca/assets/documents/Papers/paper108.pdf
58. Schindler M, Talkner P, Hänggi P (2005) Computing stationary free-surface shapes in microfluidics. submitted to J Fluid Mech http://arxiv.org/pdf/physics/0511217
59. Petersen S, Dreyer D, von Estorff O (2006) Assessment of finite and spectral element shape functions for efficient iterative simulations of interior acoustics. Computer Methods in Applied Mechanics and Engineering (in press)
60. Brinkmeier M, Nackenhorst U, Petersen S, von Estorff O (2006) A Numerical Model for the Simulation of Tire Rolling Noise. Journal of Sound and Vibration (submitted)
61. Lu S, Holst MJ, Bank RE, McCulloch AD, Michailova A (2005) 3D model of synchronous calcium signaling in ventricular myocyte, biophysical society annual meeting abstract issue