Mesh Algorithms for PDE with Sieve I: Mesh Distribution

Matthew G. Knepley
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Dmitry A. Karpeev
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

January 4, 2008

Abstract

We have developed a new programming framework, called Sieve, to support parallel numerical PDE[1] algorithms operating over distributed meshes. We have also developed a reference implementation of Sieve in C++ as a library of generic algorithms operating on distributed containers conforming to the Sieve interface. Sieve makes instances of the incidence relation, or *arrows*, the conceptual first-class objects represented in the containers. Further, generic algorithms acting on this arrow container are systematically used to provide natural geometric operations on the topology and also, through duality, on the data. Finally, coverings and duality are used to encode not only individual meshes, but all types of hierarchies underlying PDE data structures, including multigrid and mesh partitions.

In order to demonstrate the usefulness of the framework, we show how the mesh partition data can be represented and manipulated using the same fundamental mechanisms used to represent meshes. We

[1] Partial differential equation(s).

present the complete description of an algorithm to encode a mesh partition and then distribute a mesh, which is independent of the mesh dimension, element shape, or embedding. Moreover, data associated with the mesh can be similarly distributed with exactly the same algorithm. The use of a high level of abstraction within the Sieve leads to several benefits in terms of code reuse, simplicity, and extensibility. We discuss these benefits and compare our approach to other existing mesh libraries.

1 Introduction

Numerical PDE codes frequently comprise of two uneasily coexisting pieces: the mesh, describing the topology and geometry of the domain, and the functional *data* attached to the mesh representing the discretized fields and equations. The mesh data structure typically reflects the representation used by the mesh generator and carries the embedded geometic information. While this arrangement is natural from the point of view of mesh generation and exists in the best of such packages (e.g., [17]), it is frequently foreign to the process of solving equations on the generated mesh.

At the same time, the functional data closely reflect the linear algebraic structure of the computational kernels ultimately used to solve the equations; here the natural geometric structure of the equations, which reflects the mesh connectivity in the coupling between the degrees of freedom, is sacrificed to the rigid constraints of the solver. In particular, the most natural geometric operation of a restriction of a field to a local neighborhood entails tedious and error-prone index manipulation.

In response to this state of affairs a number of efforts arose addressing the fundamental issues of interaction between the topology, the functional data and algorithms. We note the MOAB project [20, 19, 7] and the TSTT/ITAPS SciDAC projects [7, 16, 3], the libMesh project [8], the GrAL project [4], to name just a few. Sieve shares many features with these projects, but GrAL is the closest to it in spirit. Although each of these projects addresses some of the issues outlined above, we feel that there is room for another approach.

Our Sieve framework, is a collection of interfaces and algorithms for manipulating geometric data. The design may be summarized by considering three constructions. First, data in Sieve are indexed by the underlying geometric elements, such as mesh cells, rather than by some artificial global order. Further, the local traversal of the data is based on the connectivity of the geometric elements. For example, Sieve provides operations that, given a mesh cell, traverse all the data on its interior, its boundary, or its closure. Typical operations on a Sieve are shown in Table 1 and described in greater detail in Section 2.1. In the table, topological mesh elements, such as vertices, edges, and so on, are refered to as abstract *points* [2], and the adjacency relation between two points, such as an edge and its vertex, is refered to as *covering*: an edge is coverted by its end vertices. Notice that exactly the same operation is used to obtain edges adjacent to a face as faces adjacent to a cell, without even a lurking dimension parameter. This is the *key* to enabling dimension-independent programming for PDE algorithms.

Second, the global topology is divided into a chain of local topologies with an overlap structure relating them to each other. The overlap is encoded using the Sieve data structure again, this time containing arrows relating points in different local topologies. The data values over each local piece are manipulated using the local connectivity, and each local piece may associate different data to the same global element. The crucial ingredient here is the operation of assembling the chain of local data collections into a consistent whole over the global topology.

Third, the covering arrows can carry additional information, controlling the way in which the data from the covering points are assembled onto the covered points. For example, orientation information can be encoded on the arrows to dictate an order for data returned over an element closure. More sophisticated operations are also possible, such as linear combinations which enable coordinate transformations, or the projection and interpolation necessary for multigrid algorithms. This is the central motivation behind the arrow-centric interface.

Emphasis on the covering idea stems directly from the cell complex construction in algebraic topology. We have abstracted it along the lines of category theory, with its emphasis on arrows, or morphisms, as the organizing principle. The analogy runs deeper, however, because in PDE applications meshes do not exist for their own sake, but to support geometrically structured information. The geometric structure of these data manifests itself through duality between topogical operations, such as *closure* of a mesh element, and analytical operations, such as the *restriction* of a field to a

[2] Our *points* correspond to geometric *entities* in some other approaches like MOAB or ITAPS

closed neighborhood of the element. Formally this can be seen as a reversal of arrows in a suitable category. At the practical level, this motivates the arrow-centric point of view, which allows us to load the arrows with the data (e.g., coordinate transformation parameters) making the dualization between covering and restriction possible.

The arrow-centric point of view also distinguishes our approach from similar projects such as [4]. In addition, it is different from the concept of a flexible database of geometric entities underlying the MOAB and TSTT/ITAPS methodologies (see e.g., [20] and [16]). Sieve can be thought of as a database, but one that limits the flexibility by insisting on the arrow-centric structure of the input and output and a small basic query interface optimized to reveal the covers of individual elements. This provides a compact conceptual universe shifting the flexibility to the generic algorithms enabled by a well-circumscribed container interface.

Although other compact intefaces based on a similar notion *adjacency* exist, we feel that Sieve's interface and the notion of a covering better capture the essense of the geometric notions underlying meshes, rather than mapping them onto a database-inspired language. Moreover, these adjacency queries often carry outside information, such as dimension or shape tags, which is superfluous in the Sieve interface and limits the opportunity for dimension independent programming. These geometric notions are so universal that the systematic use of covering notions is possible at all levels of hierarchy underlying PDE computation. For example, the notion of covering is used to record relations between vertices, edges and cells of other dimensions in a sieve. No separate relation is used to encode "side" adjacencies, such as "neighbor" relations between cells of the same dimension, as is done in GrAL.

In fact, the points of a sieve are not a priori interpreted as elements of different dimensions and covering can be used to encode *overlap* relations in multiple non-conforming meshes, multigrid hierarchies, or even identification of cells residing on multiple processors. Contrast this, for example, with the multiple notions employed by ITAPS to describe meshes: meshes, submeshes, mesh entities, mesh entity sets and parallel mesh decompositions. While the relations between all these concepts are of essentially similar nature, this unity is not apparent in the interface, inhibiting reuse and hindering analysis of the data structures, their capabilities and their complexity.

Undoubtedly, other approaches may be more appropriate in other computational domains. For instance, different data structures may be more appropriate for mesh generation, where very different types of queries, modifications and data need to be associated with the mesh. Partitioning algorithms may also require different data access patterns to ensure efficiency and scalability. Sieve does not pretend to address those concerns. Instead, we try to focus on the demands of numerical PDE algorithms that revolve around the idea of a field defined over a geometry. Different PDE problems use different fields and even different numbers of fields with different discretizations. The need for substantial flexibility in dealing with a broad class of PDE problems and their geometric nature are the main criterion for the admission into the Sieve interface.

In this publication we focus on the reuse of the basic covering notions at different levels of data hierarchy. In particular, the division of the topology into pieces and assembly over an overlap is among the fundamental notions of PDE analysis, numerical or otherwise. It is the essence of the domain decomposition method and can be used in parallel or serial settings, or both. In this paper we focus on this decomposition/assembly aspect of Sieve and present its capabilities with a fundamental example of this kind — the distribution of a mesh onto a collection of processors. It is a ubiquitous operation in parallel PDE simulation and a necessary first step in constructing the full distributed problem. Moreover mesh distribution makes for an excellent pedagogical problem, illustrating the powerful simplicity of the Sieve construction. The Sieve interface allows PDE algorithms, operating over data distributed over a mesh, to be phrased without reference to the dimension, layout, element shape, or embedding of the mesh. We illustrate this with the example of distribution of a mesh and associated data fields over it. The same simple algorithm will be used to distribute an arbitrary mesh, as well as fields of arbitrary data layout.

We discuss not only the existing code for the Sieve library but also the concepts that underlie its design and implementation. These two may not be in complete agreement, as the code continues to evolve. We use the `keyboard` font to indicate both existing library interfaces and proposed developments that more closely relate to our design concepts. Furthermore, early implementations may not be optimal from the point of view of runtime and storage complexity as we resist premature optimizations in favor of refining the interface. Nonetheless, our reference implementation is fully functional, parallel and has early users among real applications [21, 15]. This implementation verifies the viability and the consistency of the interface, but does not preclude more efficient implementations better suited to particular uses. The added value of the interface comes in the enabling of generic algorithms, which operate on the interface and are independent of the underlying implementation. In this publication we illustrate some of these fundamental algorithms.

The rest of the paper is organized as follows. In Section 2 we introduce the basic notions and algorithms of the Sieve framework, which are then seen in action in Section 3 where the algorithms for mesh distribution and redistribution in a parallel setting are discussed. Section 4 contains specific examples of mesh distribution and Section 5 concludes the paper.

2 Sieve Framework

Sieve can be viewed as a library of parallel containers and algorithms that extends the standard container collection (e.g., the Standard Template Library of C++ and BOOST libraries). The extensions are simple but provide the crucial functionality and introduce what is, in our view, a very useful semantics. Throughout this paper we freely use the modern terminology of generic programming, in particular the idea of a *concept*, which is an interface that a class must implement to be usable by templated algorithms or methods.

Our fundamental concept is that of a `Map`, which we understand in the multivalued sense as an assignment of a `sequence` of *points* in the range to each of the points in the domain. A `sequence` is an immutable ordered collection of points that can be traversed from the `begin` element to the `end`. Typically a `sequence` has no repetitions, and we assume such *set* semantics of sequences unless explicitly noted otherwise.

A `sequence` is a basic input and output type of most Sieve operations, and the basic operation acting on sequences is called `restrict`. In particular, a `Map` can be `restrict`ed to a point or a `sequence` in the domain, producing the corresponding `sequence` in the range. `Map` objects can be updated in various ways. At the minimum we require that a `Map` implement a `set` operation that assigns a `sequence` to a given domain point. Subsequent `restrict` calls may return a `sequence` reordered in an implementation-dependent way.

2.1 Basic containers

`Sieve` extends the basic `Map` concept in several ways. First, it allows bidirectional mappings. Hence we can map points in the range, called the `cap`, to the points in the domain, called the base. This mapping is called the support, while the base-to-cap mapping is called the cone.

Second, the resulting sequence actually contains not the image points but arrows. An arrow responds to source and target calls, returning respectively the cap and base points of the arrow. Thus, an arrow not only abstracts the notion of a pair of points related by the map but also allows the attachment of nearly arbitrary "payload", a capability useful for local traversals.

One can picture a Sieve as a bipartite graph with the cap above the base and the arrows pointing downward (e.g., Fig. 1). The containers are not constrained by the type of point and arrow objects, so Sieve must be understood as a library of *meta-objects* and *meta-algorithms* (a template library in the C++ notation), which generates appropriate code upon instantiation of basis objects. We primarily have the C++ setting in mind, although appropriate Python and C bindings have been provided in our reference implementation.

A Sieve can be made into a Map in two different ways, by identifying either cone or support with restrict. Each can be done with a simple adapter class and allows all the basic Map algorithms to be applied to Sieve objects as well as their descendants, such as Sieves.

The Sieve also extends Map with capabilities of more geometric character. It allows the taking of a transitive closure of cone to obtain the topological closure of a point familiar from the cell complex theory [10, 1]. Here arrows are interpreted as the incidence relations between points, which represent the cells. Likewise, iterated supports result in the star of a point. The meet(p,q) lattice operation returns the smallest sequence of points whose removal would render closure(p) and closure(q) disjoint. The join(p,q) operation is the analogue for star(p) and star(q). Note that all these operations actually return arrow sequences, but by default we extract either the source or the target, a strategy that aids in the definition of transitive closures and simplifies programming.

Fig. 1 illustrates how mesh topology can be represented as a Sieve object. The arrows indicate covering or incidence relations between triangles, edges, and vertices of a simple simplicial mesh. Sieve operations allow one to navigate through the mesh topology and carry out the traversals needed to use the mesh. We illustrate some common Sieve operations on the mesh from Fig. 1 in Table 2.

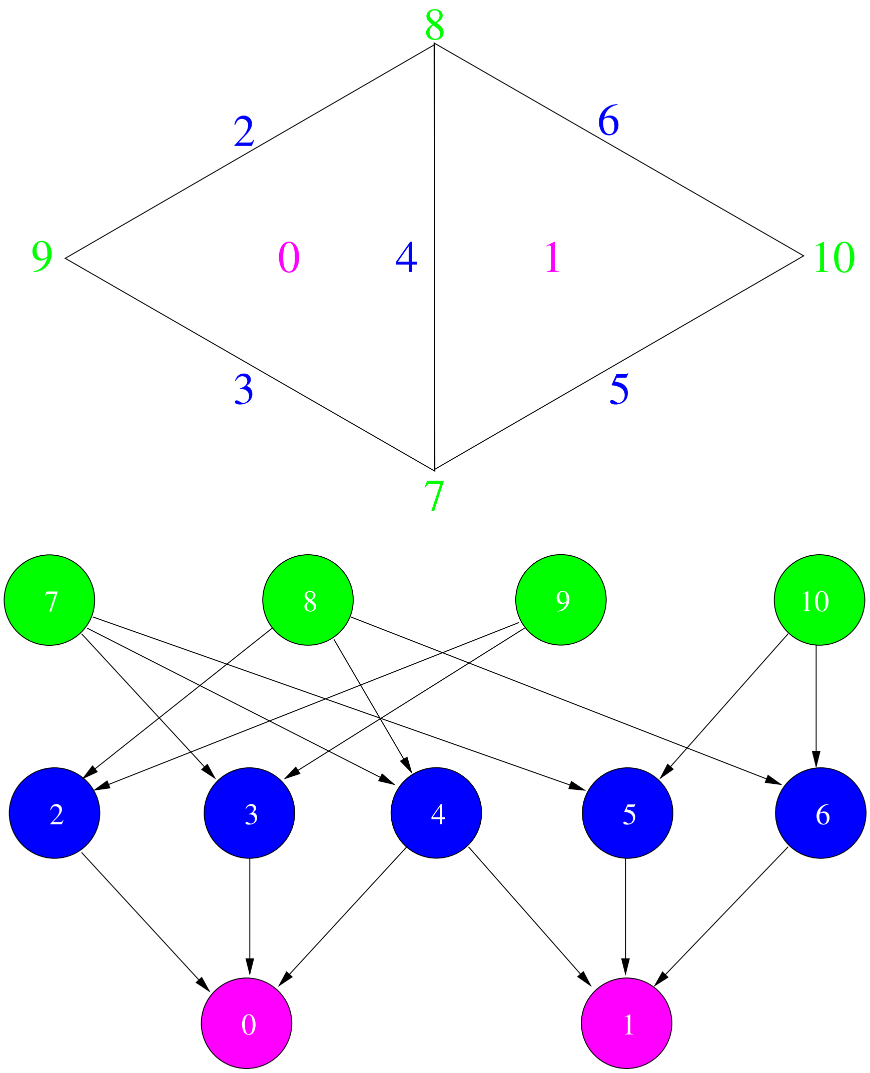


Figure 1: A simple mesh and its `Sieve` representation.

cone(p)	sequence of points covering a given point p
closure(p)	transitive closure of cone
support(p)	sequence of points covered by a given point p
star(p)	transitive closure of support
meet(p,q)	minimal separator of closure(p) and closure(q)
join(p,q)	minimal separator of star(p) and star(q)

Table 1: Typical operations on a Sieve.

cone(0)	{2, 3, 4}
support(4)	{0, 1}
closure(1)	{4, 5, 6, 7, 10, 8}
star(8)	{2, 4, 6, 0, 1}
meet(0,1)	{4}
join(2,4)	{0}
join(2,5)	{}

Table 2: Results of typical operations on the Sieve from Fig. 1

2.2 Data Definition and Assembly

Sieves are designed to represent relations between geometric entities, represented by points. They can also be used to attach data directly to arrows, but not to points, since points may be duplicated in different arrows. A Map, however, can be used effectively to lay out data over points. It defines a sequence-valued function over the implicitly defined domain set. In this case the domain carries no geometric structure, and most data algorithms rely on this minimal Map concept.

2.2.1 Sections

If a Map is combined with a Sieve, it allows more sophisticated data traversals such as restrictClosure or restrictStar. These essentially are algorithms operating on (Map, Sieve) pairs and traversing the closure or star point in the Sieve and within each the data values in the Map. Analogous traversals based on meet, join, or other geometric information encoded in Sieve can be implemented in a straightforward manner. The concept resulting from this combination is called a Section, by analogy with the geometrical notion of a section of a fiber bundle. Here the Sieve plays the role of the base space, organizing the points over which the mapping representing the section is defined. We have found Sections most useful in implementating finite element discretizations of PDE problems. These applications of Section functionality are detailed in an upcoming publication [14].

A particular implementation of Map and Section concepts ensures contiguous storage for the values. We mention it because of its importance for high-performance parallel computing with Sieve. In this implementation a Map class uses another Map internally that maps domain points to offsets into a contiguous storage array. This allows Sieve to interface with parallel linear and nonlinear solver packages by identifying Map with the vector from that package. We have done this for the PETSc [2] package. The internal Map is sometimes called the *atlas* of that Section. The analogous geometric object is the local trivialization of a fiber bundle that organizes the space of values over a domain neighborhood (see, e.g., [18]).

We observe that Sections and Sieves are in duality. This duality is expressed by the relation of the restrict operation on a Section to the cone operation in a Sieve. Corresponding to closure is the traversal of the Section data implemented by restrictClosure. In this way, to any Sieve traversal, there corresponds a traversal of the corresponding Section. Pictured another way, the covering arrows in a Sieve may be reversed to indicate restriction. This duality will arise again when we picture the dual of a given mesh in Section 3.1.

2.2.2 Overlap and Delta

In order to ensure efficient local manipulation of the data within a Map or a Section, the global geometry is divided into manageable pieces, over which the Maps are defined. In the context of PDE problems, the chain of subdomains typically represents local meshes that cover the whole domain. The dual chain, or a cochain, of Maps represents appropriate restrictions of the data to each subdomain. For PDEs, the cochain comprises local fields defined over submeshes.

The covering of the domain by subdomains is encoded by an Overlap object. It can be implemented by a Sieve, whose arrows connect the points in different subdomains that cover each other. Strictly speaking, Overlap arrows relate pairs (domain, domain_point). Alternatively, we can view Overlap itself as a chain of Sieves indexed by nonempty overlaps of the subdomains in the original chain. This better reflects the locality of likely Overlap traversal patterns: for a given chain domain, all points and their covers from other subdomains are examined.

An Overlap is a many-to-many relation. In the case of meshes this allows for nonconforming overlapping submeshes. However, the essential uses of Overlap are evident even in the simplest case representing conforming subdomain meshes treated in detail in the example below. Fig. 2 illustrates the Overlap corresponding to a conforming mesh chain resulting from partitioning of the mesh in Fig. 1. Here the Overlap is viewed as a chain of Sieves, and the local mesh point indices differ from the corresponding global indices in Fig. 1. This configuration emphasizes the fact that no global numbering scheme is imposed across a chain and the global connectivity is always encoded in the Overlap. In the present case, this is simply a one-to-one identification relation. Moreover, many overlap representations are possible; the one presented above, while straightforward, differs from that shown in Section 3.2.

The values in different Maps of a cochain are related as well. The relation among them reflects the overlap relation among the points in the underlying subdomain chain. The nature of the relationship between values varies according to the problem. For example, for conforming meshes (e.g., Fig. 2) the Overlap is a one-to-one relation between identified elements of different subdomain meshes. In this case, the Map values over the same mesh element in different domains can be duplicates, as in finite differences, or partial values that have to be added to obtain the unique global value, as in finite element methods. In either case the number of values over a shared mesh element must be the same in the cooverlapping Maps. Sometimes this number is referred to as the *fiber dimension*, by analogy with fiber bundles.

Vertex coordinates are an example of a cochain whose values are simply duplicated in different local maps, as shown in Section 3.2. In the case of nonconforming subdomain meshes, Overlap is a many-to-many relation, and Map values over overlapping points can be related by a nontrivial transformation or a relation. They can also be different in number. All of this information — fiber dimensions over overlapping points, the details of the data transformations, and other necessary information — is encoded in a Delta class.

A Delta object can be viewed as a cochain of maps over an Overlap chain, and is dual to the Overlap in the same way that a Section is dual to a Sieve. More important, a Delta acts on the Map cochain with domains related by

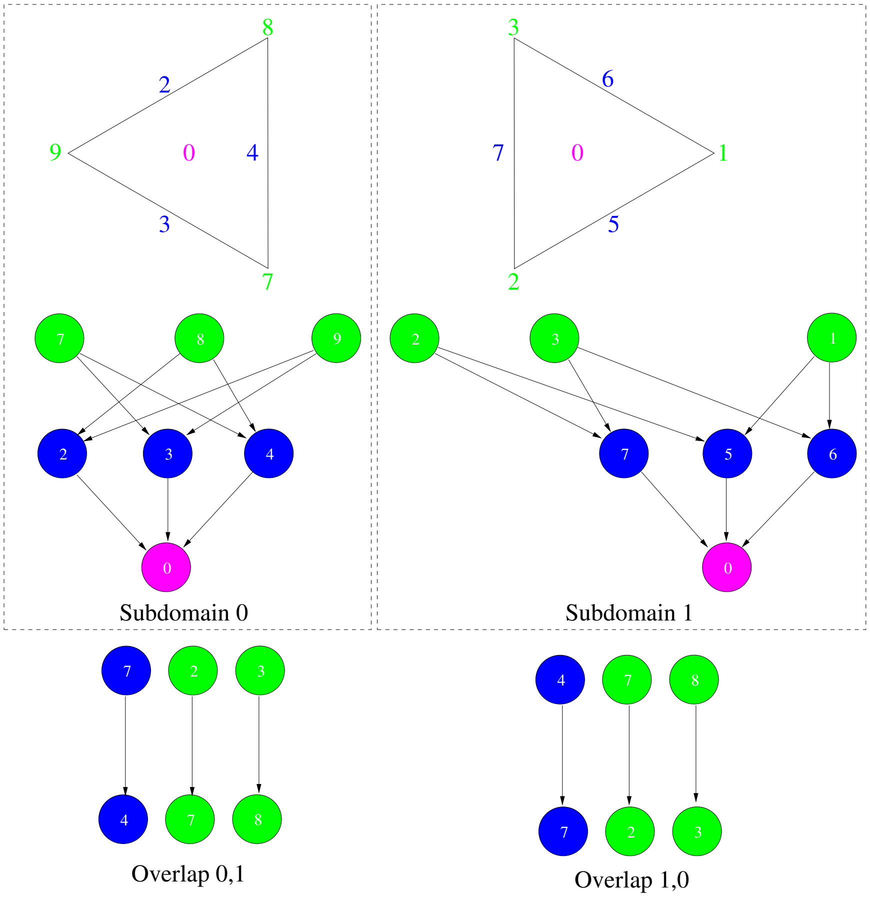


Figure 2: `Overlap` of a conforming mesh chain obtained from breaking up the mesh in Fig. 1.

the Overlap. Specifically, the Delta class defines algorithms that restrict the values from a pair of overlapping subdomains to their intersection. This fundamental operation borrowed from the *sheaf* theory (see, e.g., [5]) allows us to detect Map cochains that agree on their overlaps. Moreover (and this is a uniquely computational feature), Delta allows us to fuse the values on the overlap back into the corresponding local Maps so as to ensure that they agree on the overlap and define a valid global map. The restrict-fuse combination is a ubiquitous operation called completion, which we illustrate here in detail in the case of *distributed* Overlap and Delta. For example, in Section 3.2 we use completion to enforce the consistency of cones over points related by the overlap.

If the domain of the cochain Map carries no topology — no connectivity between the points — it is simply a set and need not be represented by a Sieve. This is the case for a pure linear algebra object, such as a PETSc Vec. However, the Overlap and Delta still contain essential information about the relationship among the subdomains and the data over them, and must be represented and constructed explicitly. In fact, a significant part of an implementation of any domain decomposition problem should be the specification of the Overlap and Delta pair, as they are at the heart of the problem.

Observe that Overlap fulfills Sieve functions at a larger scale, encoding the domain topology at the level of subdomains. In fact, Overlap can be thought of as the "superarrows" between domain "superpoints." Thus, the essential ideas of encoding topology by arrows indicating overlap between pieces of the domain is the central idea behind the Sieve interface. Likewise, Deltas act as Maps on a larger scale and can be restricted in accordance with an Overlap.

2.3 Database interpretation

The arrow-centric formalism of Sieve and the basic operations have an interpretations in terms of relational databases and the associated 'entity-relation' analyses. Indeed, Sieve points can naturally be interpreted as the rows of a table of 'entities' (both in the database sense and the sense of 'topological entity') with the point itself serving as the key. Arrows encode covering relations between points, and therefore define a natural binary database relation with the composite key consisting of the two involved points. In this scenario cones and supports have various interpretations in terms of queries against such a schema; in particular, the `cone` can be viewed as the result of a (database) join of the arrow table with the point table on the target key; the `support` is the join with the source key. More interestingly, the topological `closure` is the transitive closure of the database join applied to the arrow table; similarly for `star`. Moreover, `meet` and `join` in the topological sense cannot be formulated quite as succinctly in terms of database queries, but are very clear in terms of the geometric intuitive picture of Sieve.

This can be contrasted with the scenario, in which only point entity tables are present and the covering or incident points are stored in the entity record alongside the point key. In this case, however, arrows have no independent existence, are incapable of carrying their own ancillary information and are duplicated by each of the two related points. While in this paper we do not focus on the applications of arrow-specific data that can be attached to the arrow records for lack of space, we illustrate its utility with a brief sketch of an example.

In extracting the cone or the (topological) closure of a point, such as a hexahedron in a 3D hex mesh, it is frequently important to traverse the resulting faces, edges and points in the order determined by the orientation of the covered hex. Each face, except those on the boundary, cover two hexahedra and most edges and vertices cover several faces and edges, respectively. Each of those covering relations induces a different orientation on the face, edge or vertex. In FEM applications this results in a change of the sign of integral over the covering point. The sign, however, is not intrinsically associated with the covering point, by rather with its orientation relative to the orientation induced by the covered entity. Thus, the sign of the integral is determined by the (covering,covered) pair, that is, by the arrow. In a entity-only schema, at worst there would be no natural place for the orientation data, and at best it would make for an awkward design and potentially lead to storage duplication. More sophisticated uses of arrow-specific data include general transformation of the data attached to points upon its pullback onto the covered points (consider, for example, the restriction/prolongation multigrid operators).

To summarize, Sieve can be viewed as an interface defining a relational database with a very particular schema and a limit query set. This query set, however, allows for some operations that may be difficult to describe succinctly in the database language (topological `meet` and `join`)). Furthermore, by defining a *restricted* database of topological entities and relations, as opposed to a flexible one, Sieve potentially allows for more effective optimizations of the runtime and storage performance behind the same interface. These issues will be discussed elsewhere.

3 Mesh Distribution

Before our mesh is distributed, we must decide on a suitable partition, for which there are many excellent packages (see, e.g., [12, 13, 11]). We first construct suitable overlap `Sieves`. The points will be abstract "partitions" that represent the sets of cells in each partition, with the arrows connecting abstract partitions on different processes. The `Overlap` is used to structure the communication of `Sieve` points among processes since the algorithm operates only on `Sections`, in this case we exhibit the mesh `Sieve` as a `Section` with values in the space of points.

3.1 Dual Graph and Partition encoding

The graph partitioning algorithms in most packages, for example ParMetis and Chaco which were used for testing, require the dual to our original mesh, sometimes referred to as the element connectivity graph. These packages partition vertices of a graph, but FEM computations are best load-balanced by partitioning elements. Consider the simple mesh and its dual, shown in Fig. 3. The dual Sieve is identical to the original except that all arrows are reversed. Thus, we have an extremely simple characterization of the dual.

It is common practice to omit intermediate elements in the `Sieve`, for instance storing only cells and vertices. In this case, we may construct the dual edges on the fly by looping over all cells in the mesh, taking the `support`, and placing a dual edge for any support of the correct size (greater than or equal to the dimension is sufficient) between the two cells in the support. Note this algorithm also works in parallel because the supports will, by definition, be identical on all processes after support completion. Moreover, it is independent of the cell shape and dimension, unless the dual edges must be constructed.

The partitioner returns an assignment of cells, vertices in the dual, to partitions. This can be thought of as a `Section` over the mesh, giving the partition number for each cell. However, we will instead interpret this assignment as a `Section` over the abstract partition points taking values in the space of `Sieve` points, which can be used directly in our generic `Section`

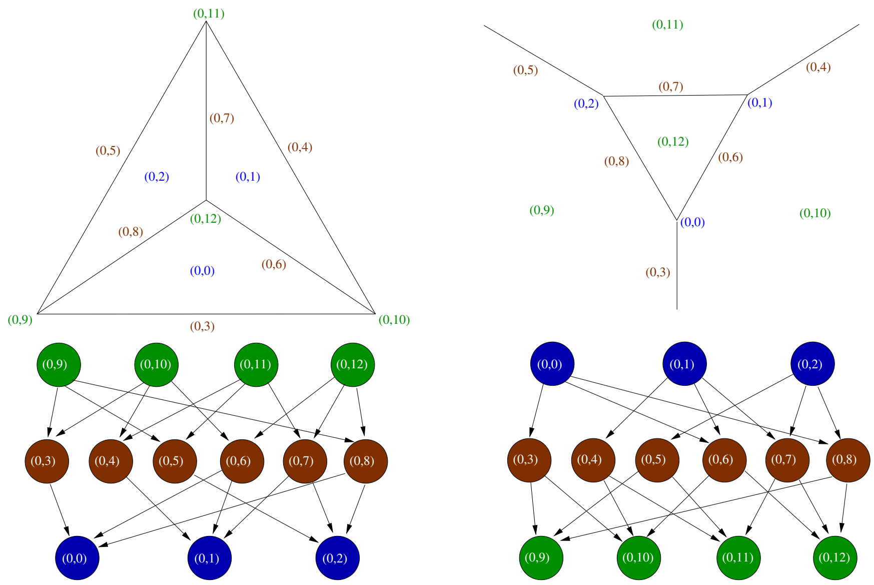


Figure 3: A simple mesh and its dual.

completion routine, described in Section 3.2.1. In fact, Sieve can generate a partition of mesh elements of any dimension, for example mesh faces in a finite volume code, using a hypergraph partitioner, such as Zoltan [6] and exactly the same distribution algorithm.

3.2 Distributing a Serial Mesh

To make sense of a finite element mesh, we must first introduce a few new classes. A `Topology` combines a sequence of `Sieves` with an `Overlap`. Our `Mesh` is modeled on the fiber bundle abstraction from topology. Analogous to a topology combined with a fiber space, a `Mesh` combines a `Topology` with a sequence of `Sections` over this topology. Thus, we may think of a `Mesh` as a `Topology` with several distinguished `Sections`, the most obvious being the vertex coordinates.

After the topology has been partitioned, we may distribute the `Mesh` in accordance with it, following the steps below:

1. Distribute the `Topology`.

2. Distribute maps associated to the topology.

3. Distribute bundle sections.

Each distribution is accomplished by forming a specific Section, and then distributing that Section in accordance with a given overlap. We call this process *section completion*, and it is responsible for all communication in the Sieve framework. Thus, we reduce parallel programming for the Sieve to defining the correct Section and Overlap, which we discuss below.

3.2.1 Section Completion

Section completion is the process of completing cones, or supports, over a given overlap. Completion means that the cone over a given point in the Overlap is sent to the Sieve containing the neighboring point, and then fused into the existing cone of that neighboring point. By default, this fusion process is just insertion, but any binary operation is allowed. For maximum flexibility, this operation is not carried out on global Sections, but rather on the restriction of a Section to the Overlap, which we term *overlap sections*. These can then be used to update the global Section.

The algorithm uses a recursive approach based on our decomposition of a Section into an atlas and data. First the atlas, also a Section, is distributed, allowing receive data sizes to be calculated. Then the data itself is sent. In this algorithm, we refer to the atlas, and its equivalent for section adapters, as a *sizer*. Here are the steps in the algorithm:

1. Create send and receive sizer overlap sections.

2. Fill send sizer section.

3. Communicate.

4. Create send and receive overlap sections.

5. Fill send section.

6. Communicate.

The recursion ends when we arrive at a ConstantSection, described in [14], which does not have to be distributed because it has the same value on every point of the domain.

3.2.2 Sieve Construction

The distribution process uses only section completion to accomplish all communication and data movement. We use adapters [9] to provide a Section interface to data, such as the partition. The PartitionSizeSection adapter can be restricted to an abstract partition point, returning the total number of sieve points in the partition (not just the those divided by the partitioner). Likewise, the PartitionSection returns the points in a partition when restricted to the partition point. When we complete this section, the points are distributed to the correct processes. All that remains is to establish the correct hierarchy among these points, which we do by establishing the correct cone for each point. The ConeSizeSection and ConeSection adapters for the Sieve return the cone size and points respectively when restricted to a point. We see here that a sieve itself can be considered a section taking values in the space of points. Thus sieve completion consists of the following:

1. Construct local mesh from partition assignment by copying.
2. Construct initial partition overlap.
3. Complete the partition section to distribute the cells.
4. Update the Overlap with the points from the overlap sections.
5. Complete the cone section to distribute remaining Sieve points.
6. Update local Sieves with cones from the overlap sections.

The final Overlap now relates the parallel Sieve to the initial serial Sieve. Note that we have used only the cone() primitive, and thus this algorithm applies equally well to meshes of any dimension, element shape, or connectivity. In fact, we could distribute an arbitrary graph without changing the algorithm.

3.3 Redistributing a Mesh

Redistributing an existing parallel mesh is identical to distributing a serial mesh in our framework. However, now the send and receive Overlaps are potentially nonempty for every process. The construction of the intermediate partition and cone Sections, as well as the section completion algorithm, remain exactly as before. Thus, our high level of abstraction has resulted in enormous savings through code reuse and reduction in complexity.

As an example, we return to the triangular mesh discussed earlier. However, we will begin with the distributed mesh shown in Fig. 4, which assigns triangles (4, 5, 6, 7) to process 0, and (0, 1, 2, 3) to process 1. The main difference in this example will be the `Overlap`, which determines the communication pattern. In Fig. 5, we see that each process will both send and receive data during the redistribution. Thus, the partition `Section` in Fig. 6 has data on both processes. Likewise, upon `completion` we can construct a Sieve `Overlap` with both send and receive portion on each process. Cone and coordinate `completion` also proceed exactly as before, except that data will flow between both processes. We arrive in the end at the redistributed mesh shown in Fig. 7. No operation other than Section `completion` itself was necessary.

4 Examples

To illustrate the distribution method, we begin with a simple square triangular mesh, shown in Fig. 8 with its corresponding `Sieve` shown in Fig. 9. We distribute this mesh onto two processes: the partitioner assigns triangles (0, 1, 2, 4) to process 0, and (3, 5, 6, 7) to process 1. In step 1, we create a local `Sieve` on process 0, shown in Fig. 10, since we began with a serial mesh.

For step 2, we identify abstract partition points on the two processes using an overlap `Sieve`, shown in Fig. 11. Since this step is crucial to an understanding of the algorithm, we will explain it in detail. Each `Overlap` is a `Sieve`, with dark circles representing abstract partition points, and light circles process ranks. The rectangles are Sieve `arrow` data, or labels, representing remote partition points. The send `Overlap` is shown for process 0, identifying the partition point 1 with the same point on process 1. The corresponding receive `Overlap` is shown for process 1. The send `Overlap` for process 1 and receive `Overlap` for process 0 are both null because we are broadcasting a serial mesh from process 0.

We now complete the partition `Section`, using the partition `Overlap`, in order to distribute the Sieve points. This `Section` is shown in Fig. 12. Not only are the four triangles in partition 1 shown, but also the six vertices. The receive overlap `Section` has a base consisting of the overlap points, in this

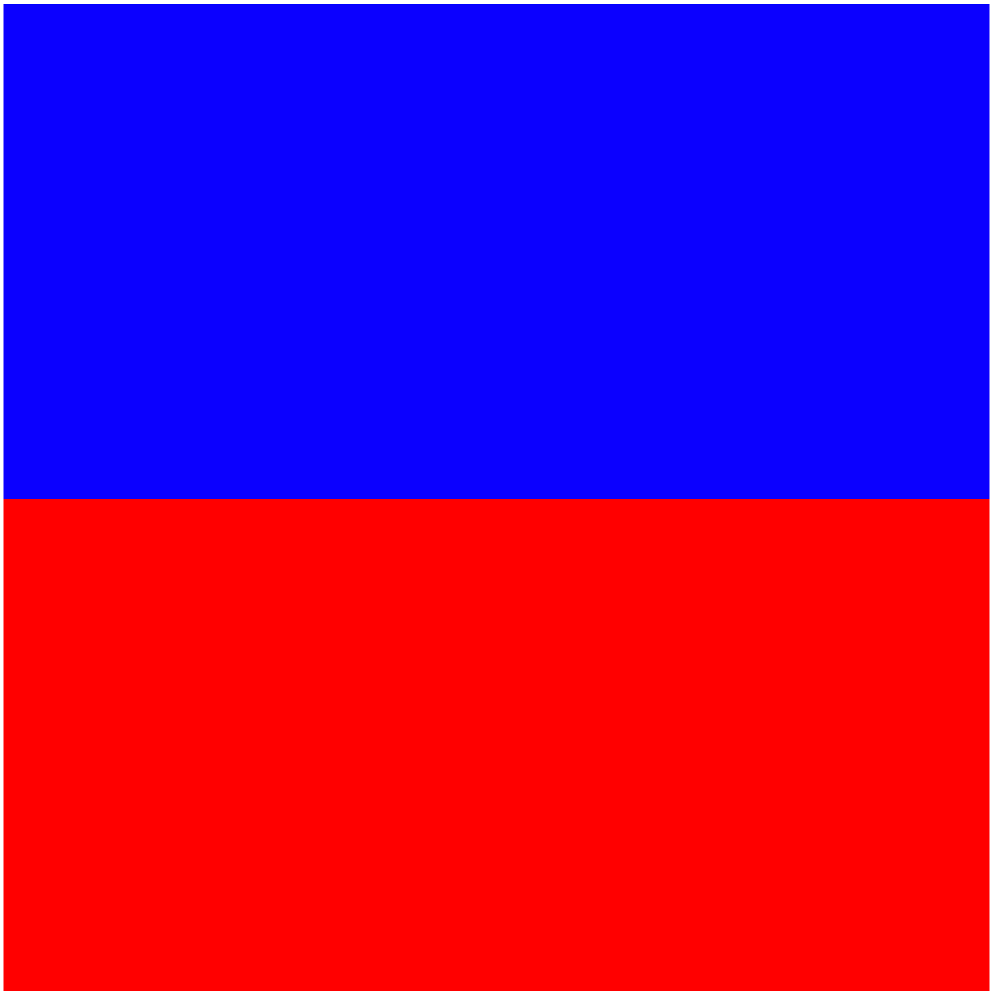


Figure 4: Initial distributed triangular mesh.

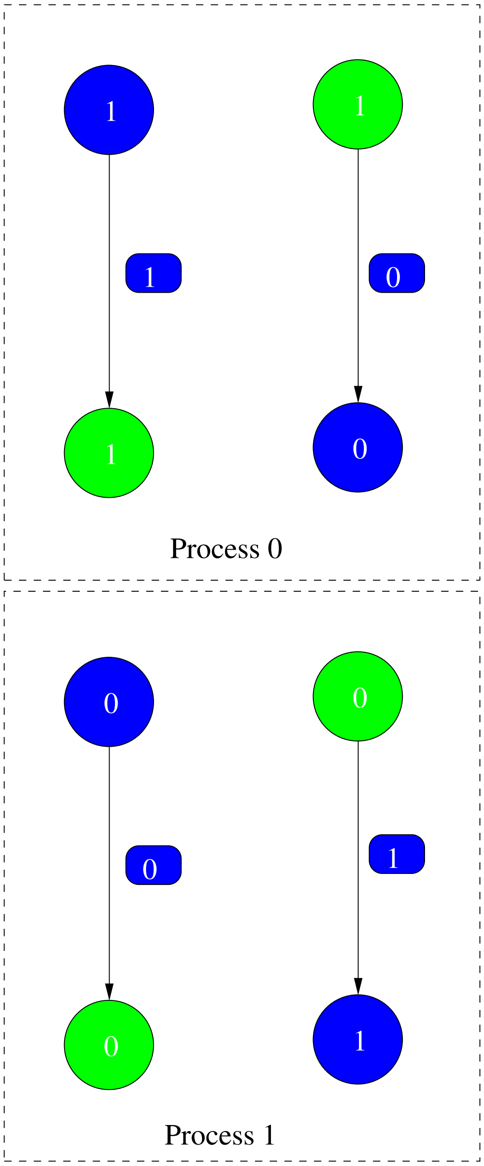


Figure 5: Partition point `Overlap`, with dark partition points, light process ranks, and `arrow` labels representing remote points. The send `Overlap` is on the left, and the receive `Overlap` on the right.

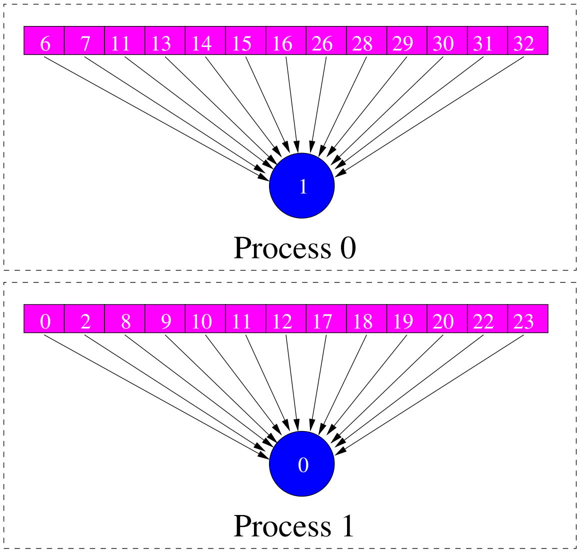


Figure 6: Partition section, with circular partition points and rectangular Sieve point data.

case partition point 1; the cap will be completed, meaning that it now has the Sieve points in the cap.

Using the receive overlap `Section` in step 4, we can update our `Overlap` with the new Sieve points just distributed to obtain the `Overlap` for Sieve points rather than partition points. The Sieve `Overlap` is shown in Fig. 13. Here identified points are the same on both processes, but this need not be the case. In step 5 we complete the cone `Section`, shown in Fig. 14, distributing the covering relation. We use the `cones` in the receive overlap `Section` to construct the distributed Sieve in Fig. 15.

After distributing the topology, we distribute any associated `Sections` for the `Mesh`. In this example, we have only a coordinate `Section`, shown in Fig. 16. Notice that while only vertices have coordinate values, the Sieve `Overlap` contains the triangular faces as well. Our algorithm is insensitive to this, as the faces merely have empty `cones` in this `Section`. We now make use of another adapter, the `Atlas`, which substitutes the number of values for the values returned by a `restrict`, which we use as the sizer for `completion`. After distribution of this `Section`, we have the result in Fig. 17. We are thus able to fully construct the distributed mesh in Fig. 18.

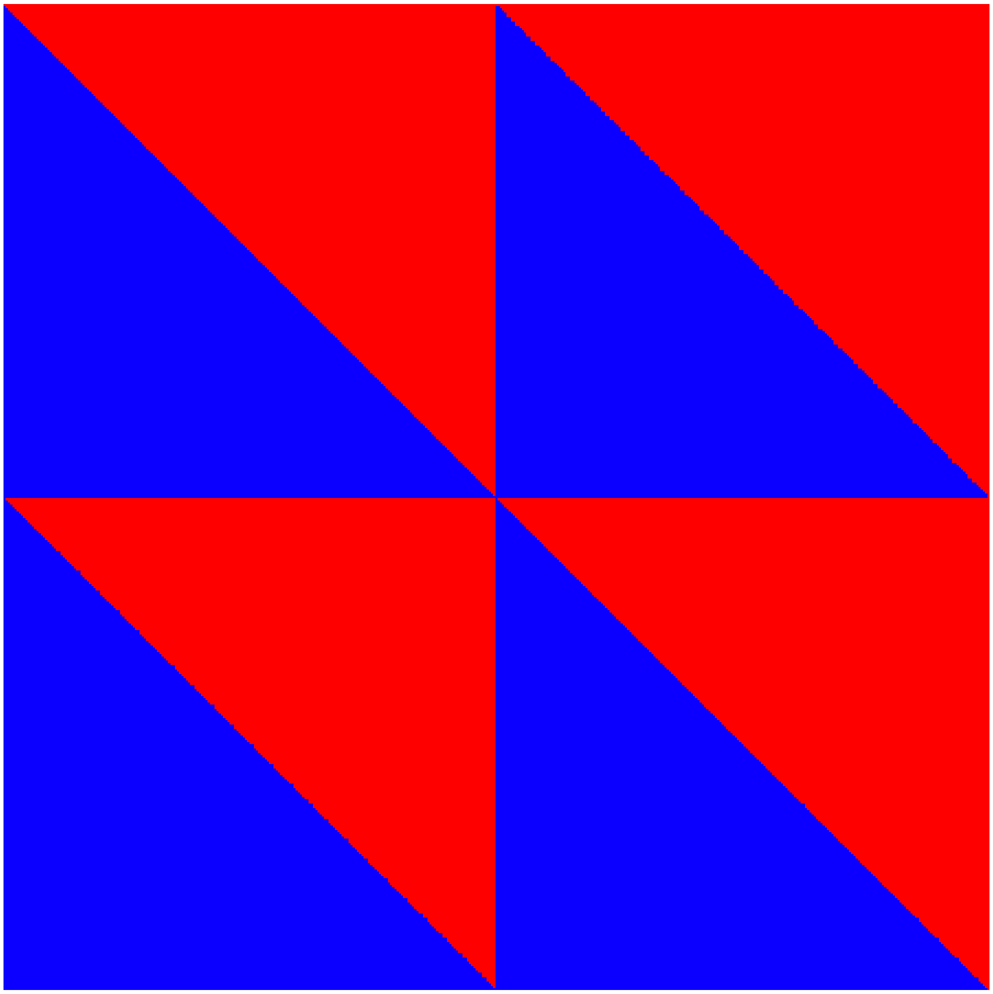


Figure 7: Redistributed triangular mesh.

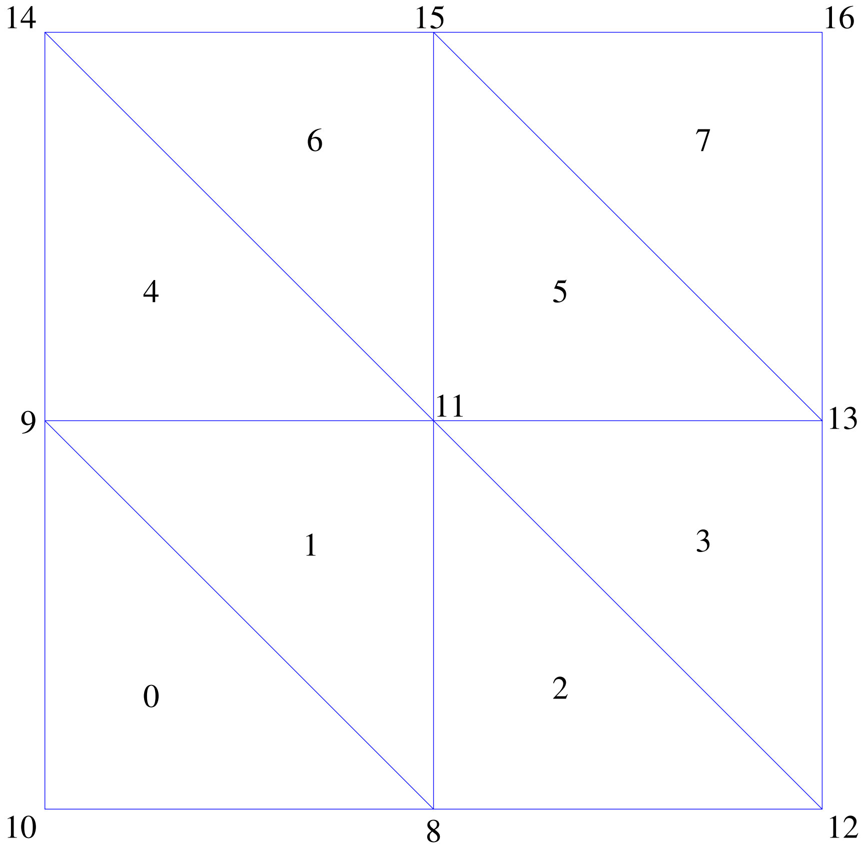


Figure 8: A simple triangular mesh.

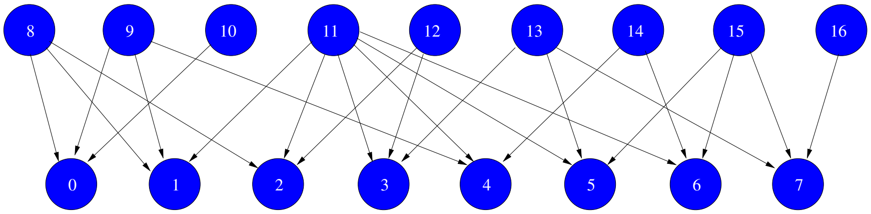


Figure 9: `Sieve` for mesh in Fig. 8.

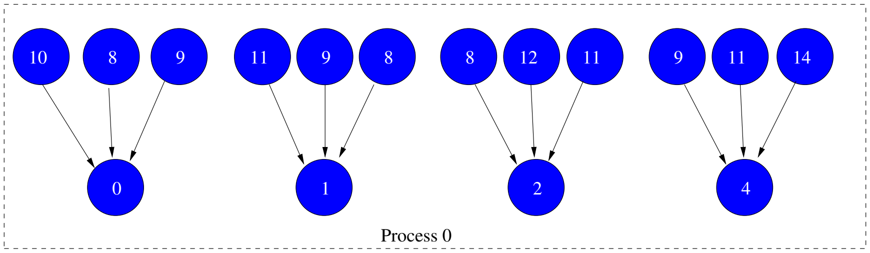


Figure 10: Initial local sieve on process 0 for mesh in Fig. 8.

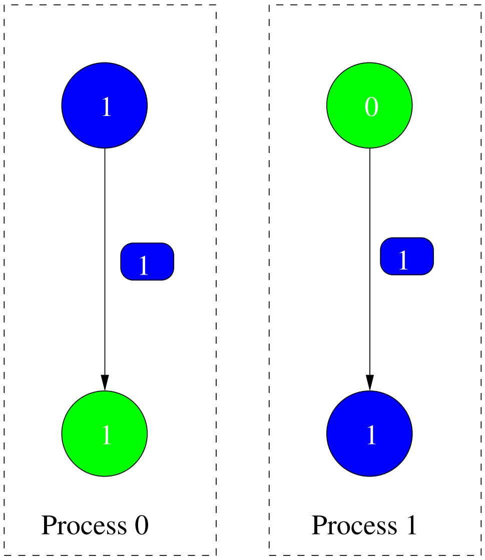


Figure 11: Partition point `Overlap`, with dark partition points, light process ranks, and `arrow` labels representing remote points.

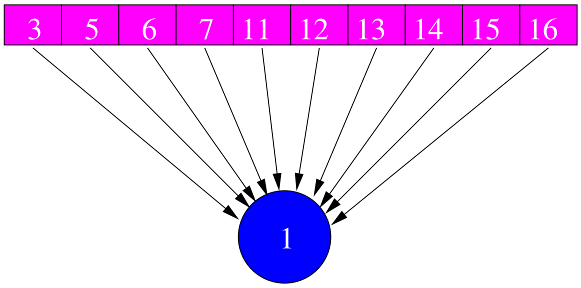


Figure 12: Partition section, with circular partition points and rectangular Sieve point data.

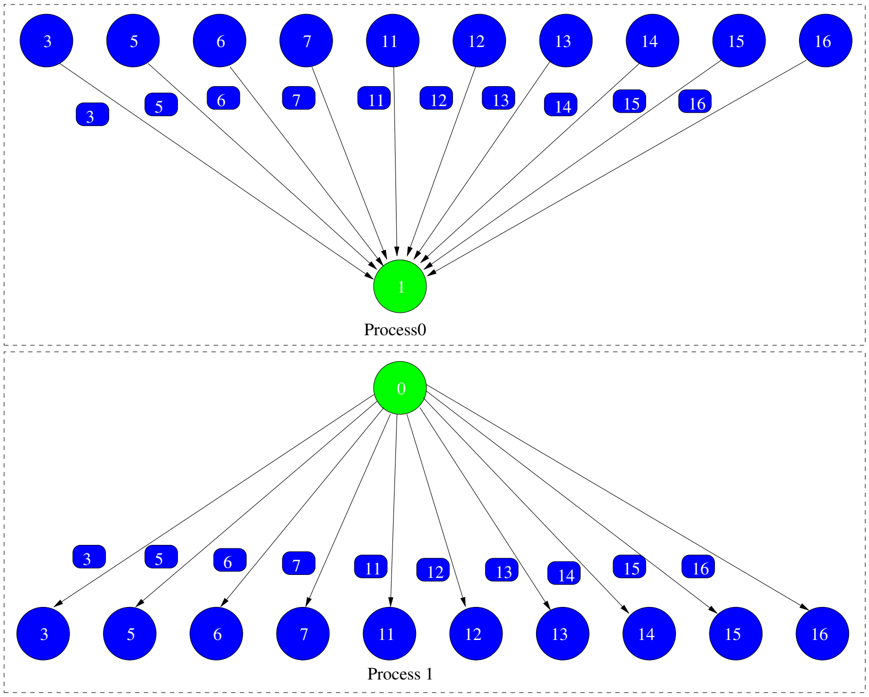


Figure 13: Sieve overlap, with Sieve points in blue, process ranks in green, and `arrow` labels representing remote sieve points.

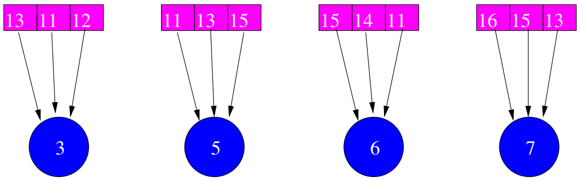


Figure 14: Cone `Section`, with circular Sieve points and rectangular `cone` point data.

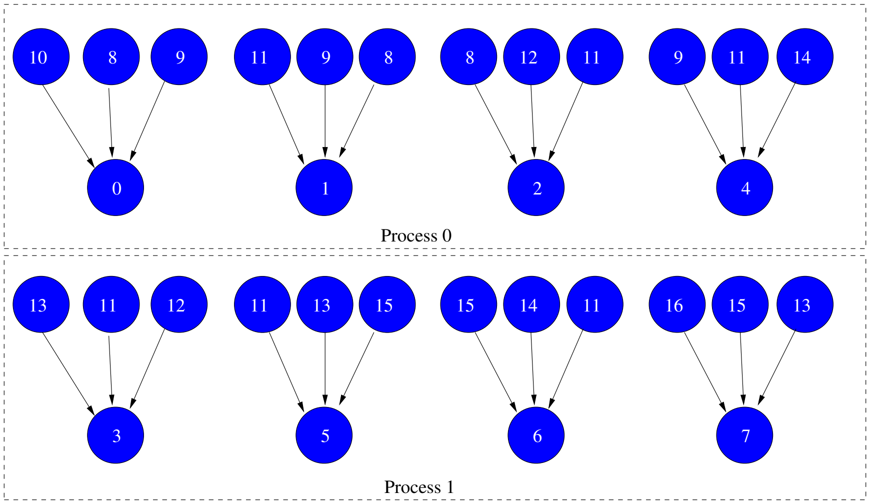


Figure 15: Distributed Sieve for mesh in Fig. 18.

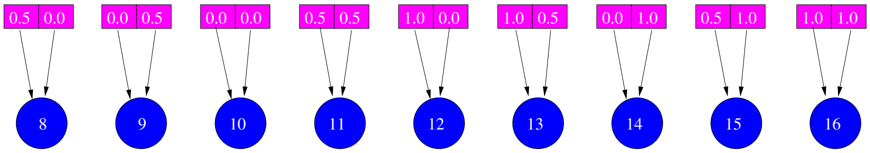


Figure 16: Coordinate `Section`, with circular Sieve points and rectangular coordinate data.

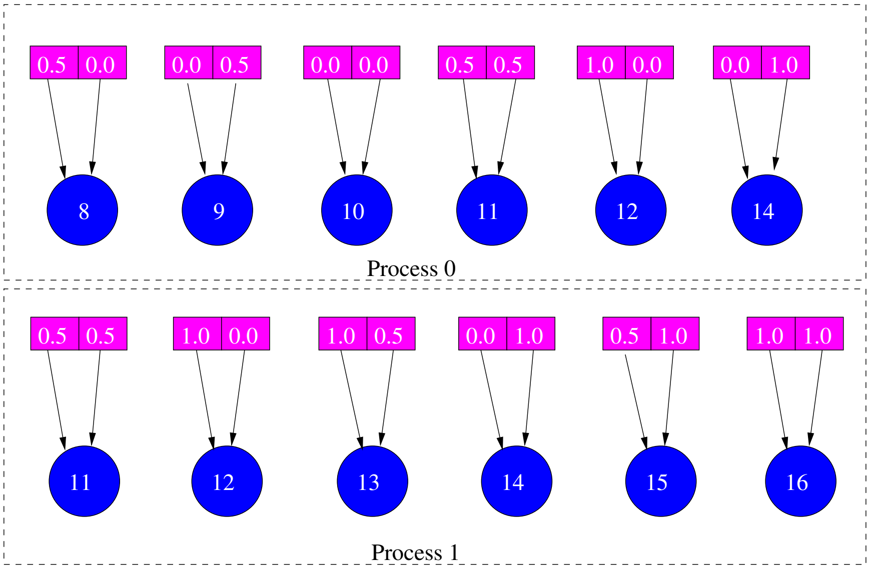


Figure 17: Distributed coordinate `Section`.

The mesh distribution method is independent of the topological dimension of the mesh, the embedding, the cell shapes, and even the type of element determining the partition. Moreover, it does not depend on the existence of intermediate mesh elements in the Sieve. We will change each of these in the next example, distributing a three-dimensional hexahedral mesh, shown in Fig. 19, by partitioning the faces. As one can see from Fig. 20, the Sieve is complicated even for this simple mesh. However, it does have recognizable structures. Notice that it is stratified by the topological dimension of the points. This is a feature of any cell complex when represented as a Sieve.

The partition `Overlap` in this case is exactly the one shown in Fig. 11; even though an edge partition was used instead of the cell partition common for finite elements, the partition `Section` in Fig. 21 looks the same although with more data. Not only is the closure of the edges included, but also their star. This is the abstract method to determine all points in a given partition. The Sieve `Overlap` after `completion` is also much larger but has exactly the same structure. In fact, all operations have exactly the same form because the section `completion` algorithm is independent of all the extraneous details in the problem. The final partitioned mesh is shown in Fig. 22, where we see that ghost cells appear automatically when we use a face partition.

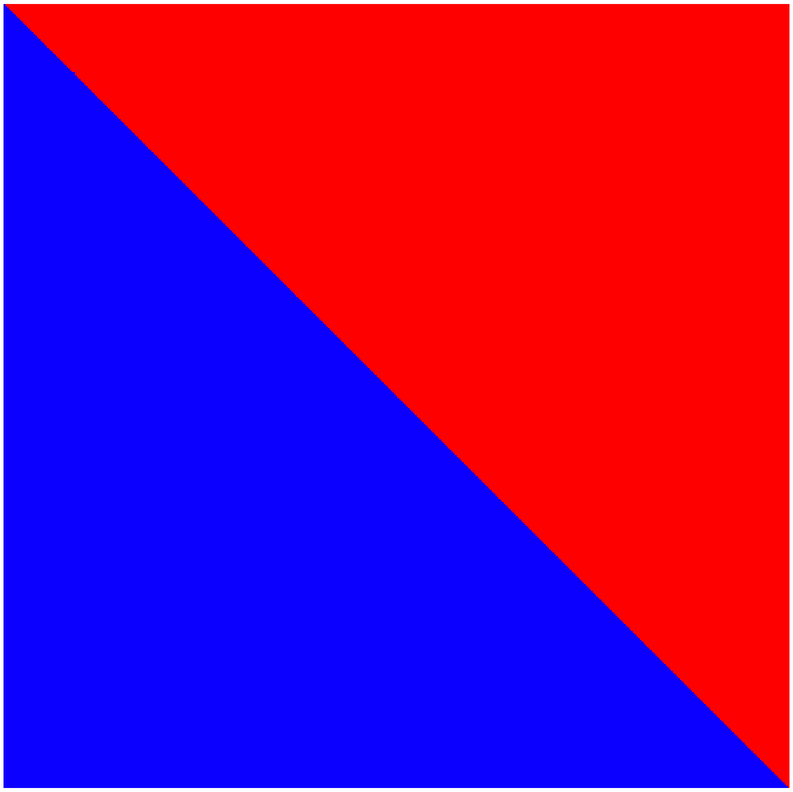


Figure 18: The distributed triangular mesh.

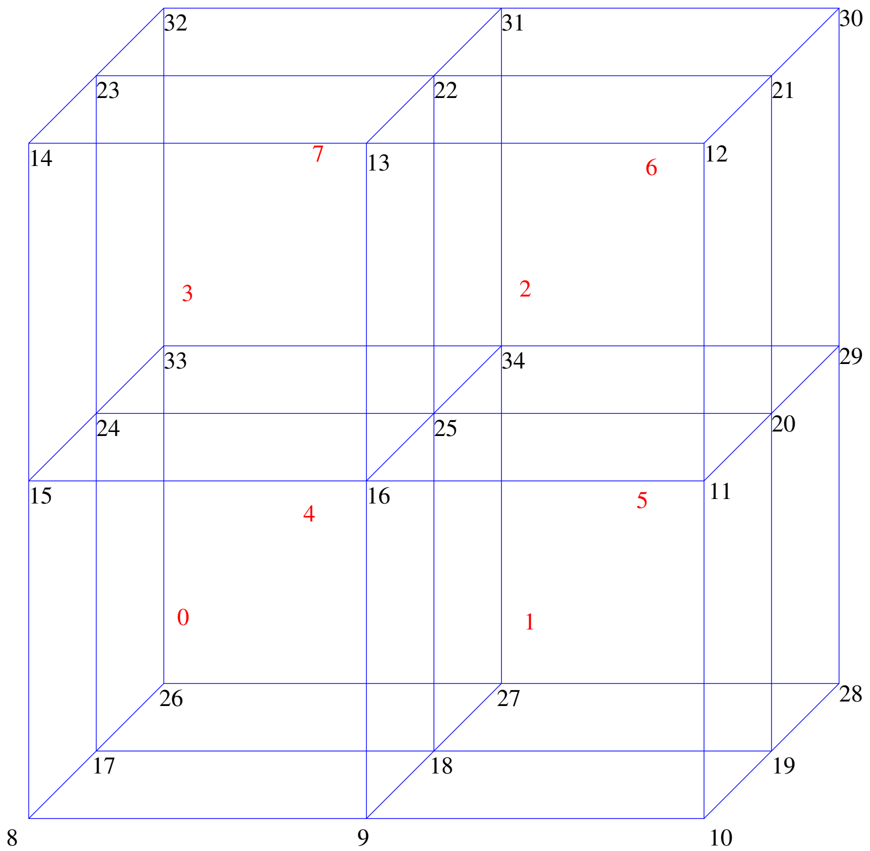


Figure 19: A simple hexahedral mesh.

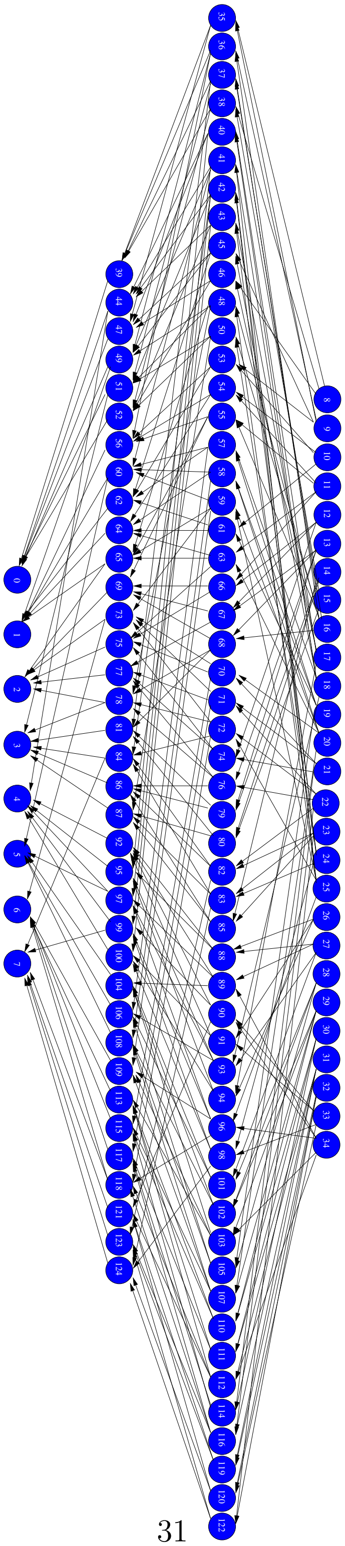


Figure 20: Sieve corresponding to the mesh in Fig. 19.

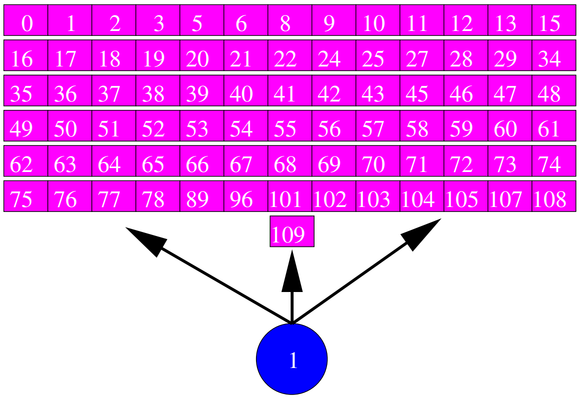


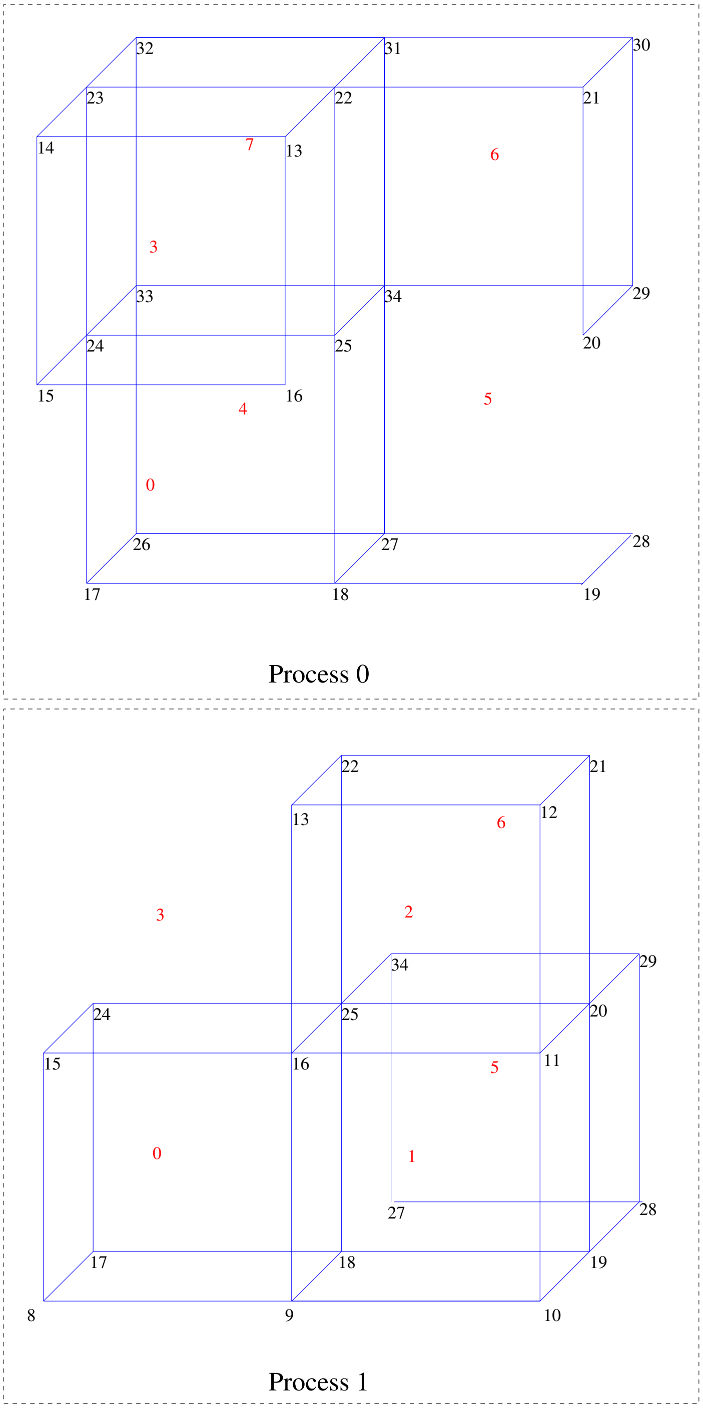
Figure 21: Partition `Section`, with circular partition points and rectangular Sieve point data.

5 Conclusions

We have presented mesh partitioning and distribution in the context of the Sieve framework in order to illustrate the power and flexibility of this approach. Since we draw no distinction between mesh elements of any shape, dimension, or geometry, we may accept a partition of any element type, such as cells or faces. Once provided with this partition and an overlap sieve, which just indicates the flow of information and is constructed automatically, the entire mesh can be distributed across processes by using a single operation, *section completion*. Thus, only a single parallel operation need be portable, verifiable, or optimized for a given architecture. Moreover, this same operation can be used to distribute data associated with the mesh, in any arbitrary configuration, according to the same partition. Thus, the high level of mathematical abstraction in the Sieve interface results in concrete benefits in terms of code reuse, simplicity, and extensibility.

Acknowledgements

The authors benefited from many useful discussions with Gary Miller and Rob Kirby. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy,



Process 0

Process 1

Figure 22: Distributed hexahedral mesh.

under Contract DE-AC02-06CH11357.

References

[1] Pavel S. Aleksandrov. *Combinatorial Topology*, volume 3. Dover, 1998.

[2] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.3.2, Argonne National Laboratory, September 2006. see `http://www.mcs.anl.gov/petsc`.

[3] M. W. Beall, J. Walsh, and Mark S. Shephard. A comparison of techniques for geometry access related to mesh generation. *Engineering With Computers*, 20(3):210–221, 2004.

[4] Guntram Berti. *Generic Software Components for Scientific Computing*. PhD thesis, TU Cottbus, 2000. http://www.math.tu-cottbus.de/ berti/diss.

[5] Glen E. Bredon. *Sheaf Theory*. Graduate Texts in Mathematics. Springer, 1997.

[6] K. Devine, E. Boman, R. Heaphy, U. Catalyurek, and R. Bisseling. Parallel hypergraph partitioning for irregular problems. *SIAM Parallel Processing for Scientific Computing*, February 2006.

[7] Ray Meyers et. al. Snl implementation of the tstt mesh interface. In *8th International conference on numerical grid generation in computational field simulations*, June 2002.

[8] Graham F. Some aspects of adaptive grid technology related to boundary and interior layers. *Journal of Computational Applied Mathematics*, 166(1):55–86, 2004.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[10] Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[11] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*, page 28, New York, 1995. ACM Press.

[12] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71–85, 1998.

[13] George Karypis et al. ParMETIS Web page, 2005. `http://www.cs.umn.edu/~karypis/metis/parmetis`.

[14] Matthew G. Knepley and Dmitry A. Karpeev. Sieve implementation. Technical Report ANL/MCS to appear, Argonne National Laboratory, January 2008.

[15] Richard C. Martineau and Ray A. Berry. The pressure-corrected ice finite element method for compressible flows on unstructured meshes. *Journal of Computational Physics*, 198(2):659–685, 2004.

[16] E. S. Seol and Mark S. Shephard. A flexible distributed mesh data structure to support parallel adaptive analysis. In *Proceedings of the 8th US National Congress on Computational Mechanics*, 2005.

[17] Jonathan R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[18] Norman Steenrod. *The Topology of Fibre Bundles. (PMS-14).* Princeton University Press, April 1999.

[19] Timothy J. Tautges. Moab-sd: Integrated structured and unstructured mesh representation. *Engineering With Computers*, 20:286–293, 2004.

[20] Timothy J. Tautges, Ray Meyers, Karl Merkley, Clint Stimpson, and Corey Ernst. Moab: A mesh-oriented database. Technical Report SAND2004-1592, Sandia National Laboratories, April 2004.

[21] Charles A. Williams, Brad Aagaard, and Matthew G. Knepley. Development of software for studying earthquakes across multiple spatial and temporal scales by coupling quasi-static and dynamic simulations. In *Eos Transactions of the AGU*. American Geophysical Union, 2005. Fall Meeting Supplemental, Abstract S53A-1072.