Flexible Representation of Computational Meshes

Matthew G. Knepley and Dmitry A. Karpeev

We propose a new representation of computational meshes in terms of a *covering* relation defined by discrete topological objects we call *sieves.* Fields over a mesh are handled locally using the notion of *refinement*, dual to covering, and are later reassembled. In this approach fields are modeled by sections of a *fiber bundle* over a sieve. This approach cleanly separates the topology of the mesh from its geometry and other value-storage mechanisms. By using these abstractions we are able to express finite element calculations using algorithms that are independent of mesh dimension, global topology, element shapes, and the finite element itself. Extensions and other applications are discussed.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Algorithm design and analysis; G.1.m [**Partial Differential Equations**]: Miscellaneous; E.2 [**Data Storage Representations**]: Composite structures

General Terms: Algorithms, Theory

Additional Key Words and Phrases: mesh, discretization, finite elements, sieve

1. INTRODUCTION

Many mesh generators are freely available [Shewchuk 2005; Si 2005], in which the interfaces presented to the user closely mirror the specific characters of these meshes (e.g., triangular or hexahedral) and the specific algorithms used in the construction (e.g., incircle tests). It has become common practice to transfer these interfaces directly into PDE simulation packages in order to represent the domain. However, such practice severely reduces the flexibility and extensibility of those packages. To rectify this situation, we must return to the underlying mathematical abstractions on which the discretization algorithms themselves are based.

An essential feature of PDEs is their locality: (1) a differential operator needs the input field values only from a neighborhood of point x to compute the output value there, and (2) locally defined fields can be extended to the larger set covered by their domains if they agree on the domain intersections. This feature is reflected in the finite element method (FEM) approach to the discretized problem: operators are assembled from local pieces operating on fields restricted to mesh elements, with the assembly performed on the intersections. The intersection structure of the local pieces determines the global data flow and provides a *natural* index into the global data objects. Here we formalize the notion of a computational domain and

Author's address: Matthew Knepley, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439.

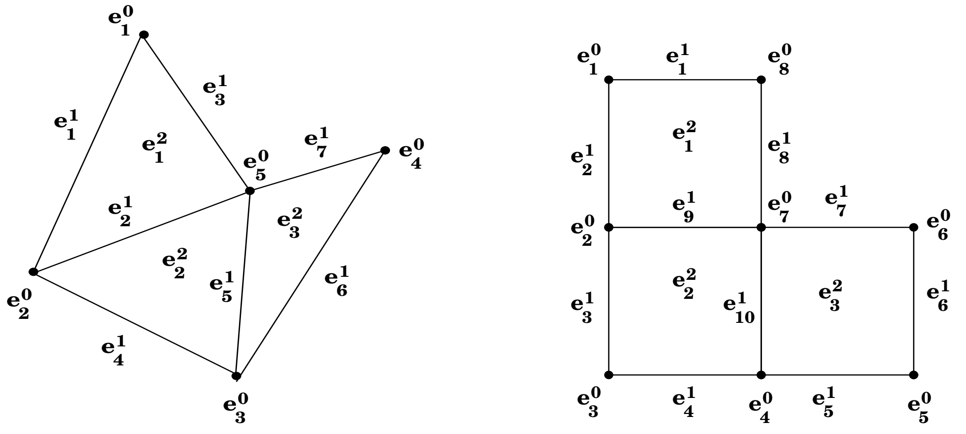

Fig. 1. Examples of simple 2D polyhedral complexes: simplicial (left) and cubic (right). Cells are labeled by their indices (subscript) within each dimension (superscript).

a field over it. The ultimate goal is the definition of discrete operators acting on fields, and the solution of discretized PDEs couched in these terms.

Moreover, in the spirit of the FENICS [Dupont et al. 2005] project in computational mathematical modeling, we focus on mathematics that describes the computation itself, rather than the solution. For instance, much effort has gone into characterizing the convergence and accuracy properties of finite element algorithms, but little work – apart from FIAT [Kirby 2004] – has sought to characterize structures capable of representing or calculating with classes of finite elements. In the way similar to the FIAT effort [Kirby], we seek to define computational structures that can represent very general hierarchies and operations over these structures from which we may construct very general algorithms.

2. COMPUTATIONAL SPACE ABSTRACTIONS

The prototypical setting where a need for computational domains arises is the application of the finite element method to the discretization of PDEs. The first requisite of the finite element method, according to the Ciarlet definition [Ciarlet 1978], is a bounded domain K with a piecewise smooth boundary. These elementary objects are images in $\mathbb{R}^d$ of some polyhedral sets assembled into a mesh by matching polyhedra faces. On the basis of this purely combinatorial information, a mesh corresponds exactly to the elements of a *polyhedral cell complex.* The best known of polyhedral cell complexes are *simplicial complexes* [Aleksandrov 1957], although we do not restrict the shape of cells a priori (see Figure 1 for examples). The basis of the complex topology is the incidence relation between adjacent cells and between the cells and their faces.

Alternatively, we can say that the mesh is *covered* by its elements. In fact, the mesh topology itself may be expressed in terms of the covering relation among its elements. For example, an edge is covered by each endpoint, and it in turn covers the faces it borders. In general, an element e' covers another element e if it is part of the boundary or $e' \in \partial e$, and all elements cover themselves. At the moment this definition appears unsatisfactory because it apparently omits the interior of the elements, but it does capture the main idea of the *covering* relation, which we want to extract and refine. It is inspired by Grothendieck's notion of a *site* [Barr and Wells 1985], as a *category* with an *étale* (or covering) topology, in which the notion of covering is axiomatized. We will examine the categorical ramifications of our approach in subsequent publications [Knepley and Karpeev]. Here we present a more traditional graph-theoretic development of these ideas.

In a sense, a cover carries all the information about the covered domain localized in its subdomains. The main advantage of such an approach to topology is that it makes natural the notion of localization useful for the definition and manipulation of fields (see Section 3). Any field defined on a domain can be restricted to a covering subdomain by composition with a covering arrow, manipulated locally, and then reassembled from the pieces supported on the elements of the cover. A recursive application of this decomposition allows one to separate a field into pieces that can be conveniently manipulated independently and then reassembled in another recursive process, or the equivalent one-step procedure, a composition of the individual assembly stages.

2.1 Sieves

To express the idea of covering in a compact and intuitive way, we introduce the *Sieve* interface. It consists of a directed acyclic graph (DAG) with covering arrows between the nodes, also known as points. In many cases, however, the primitive input and output object is a *chain*[1], or set of sieve nodes.

2.1.1 *Basic Queries.* The key operation on a sieve is the `cone`: for any sieve point `p` the output of `cone(p)` is the chain that completely covers point p – the set of all points with an arrow to `p`. By taking the cone recursively after a finite number of steps (thanks to acyclicity) we obtain the `closure(p)` of node p – the chain of all points from which p is reacheable. The dual operations of `support` and `star` are defined analogously by reversing the arrows: `support(p)` and `star(p)` respectively produce the chains of all nodes pointed to by `p` and reacheable from `p`.

A less trivial operation, the $\mathtt{meet}(\mathtt{p}_1,\mathtt{p}_2)$ of two points $\mathtt{p}_1$ and $\mathtt{p}_2$, is defined as the chain `m` of all the points from which both $\mathtt{p}_1$ and $\mathtt{p}_2$ are reacheable, and which is minimal in the sense that for any such point the paths to $\mathtt{p}_1$ and $\mathtt{p}_2$ necessarily factor through `m`. Equivalently, the meet can be described as the minimal separator of the two points – a set whose removal ensures that no point can simultaneously reach $\mathtt{p}_1$ and $\mathtt{p}_2$. The dual of `meet` is the `join` of two points.[2] Together `meet` and `join` make the set of sieve chains into a lattice. Alternatively, a lattice structure could be introduced on the set of sieve's nodes by adjoining (perhaps implicitly) special nodes: an `emptyNode`, covering each of the root nodes (nodes that, in the absence of special nodes, have zero in-degree), and a `totalSieve` node, covered by

[1]Despite what the name might suggest, no extra relations between the elements of the chain are presumed. From the point of view of combinatorial topology we are dealing with *chains mod 2* or chains with coefficients in the two-point field F_2. However, we shall not use the algebraic structure or homology in this publication.
[2]The obvious analogy with the categorical notions of *product* and *coproduct* will be clarified in our forthcoming paper [Knepley and Karpeev].

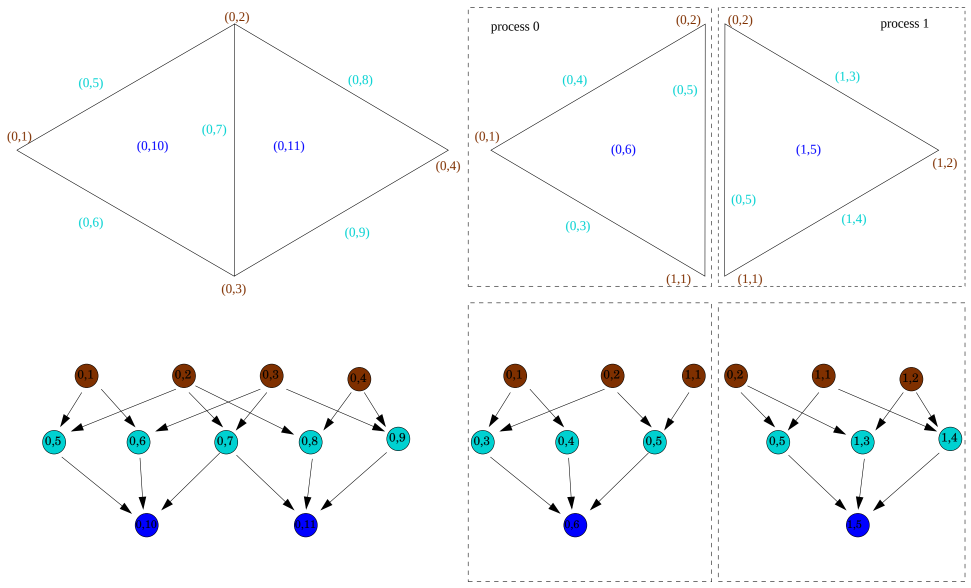

Fig. 2. Illustration of mesh representation using a sieve.

all leaf nodes (nodes that, in the absence of the special nodes, have zero out-degree). This, however, in many cases provides no information on the relationship between the nodes within a sieve. On the other hand, using chains as the primitive objects the special objects are simply represented by an empty chain as well as the chain of all leaves.

As we see, it is really the chains that play the principal role in all Sieve constructions, nodes being the special case of a singleton chain. All of the above methods, such as `meet, join`, are defined on chains simply by taking the unions of the pointwise results. In the case of `meet` and `join` the result is the union of all pairwise operations with one point from each of the two chains. We extend the notion of covering from nodes to chains by saying that a chain $\mathtt{c}'$ *covers* another chain `c` iff each node of $\mathtt{c}'$ is itself a node of `c` or covers some node of `c`. More strongly, we say that $\mathtt{c}'$ *completely covers* `c` (denoted $\mathtt{c}' \twoheadrightarrow \mathtt{c}$) iff, in addition to covering `c`, whenever $\mathtt{c}'$ omits a node p of `c`, it contains $\mathtt{cone}(p)$.

The methods described above are sufficient to express all of the mesh topology semantics. In particular, interpreting a sieve as a cell complex with the arrows indicating face inclusion, the `closure` and `star` operations correspond exactly to the same operations on the complex [Aleksandrov 1957]. We illustrate the use of sieves in representing meshes in Figure 2.

2.1.2 *Parallel Queries.* As sieves are inherently parallel objects, all of their nodes are labeled with integer (`prefix, index`) pairs to allow for independent insertion of nodes on different processes without fear of label collision. For example, the processor rank can be used as the prefix, although arbitrary integer prefixes are allowed and identically labeled nodes on different processes are treated as iden-

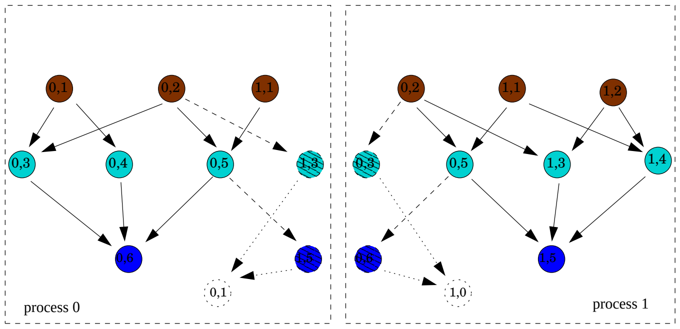

Fig. 3. Illustration of support completion (dashed) and the corresponding footprint (dotted) of a mesh sieve.

tical. At the same time, the same node on different processes may have distinct cone or support sets so that in general each process possesses only partial information about the sieve – its `localSieve` – which is a subsieve of the `totalSieve`. In fact, the ability to *complete* the local picture on any process is, in our view, the most powerful feature of sieves. Such a *completion* operation obtains the remote portion of the cone (`coneCompletion`) or the support (`supportCompletion`) over each locally-held node and stores it in a separate completion sieve. If necessary the completion can be added to the original sieve using the `add` method, forming a new sieve with the node and arrow sets formed by the unions of the node and arrow sets respectively of the added sieves. It is important to note that even after completion the local sieve may contain only partial information about the total sieve, as illustrated in Figure 3.

At this point it may be useful to introduce the notion of a *base* and *cap* of a sieve as the sets of nodes of nonzero in-degress and out-degrees respectively. Intuitively, the base can be thought of as the domain of incoming arrows and the cap as the domain of outgoing arrows, and the sieve may be conveniently pictured as a bi-partite graph with the cap lined above the base and the two levels joined by arrows. Since the intersection of the base and the cap is non-empty in general, a sieve is not a bona fide bi-partite graph, unless the occurences of the same node in the base and the cap are distinguished as separate nodes. It can be a useful way of viewing sieves, as well as a source of interface and implementation simplifications. The methods returning the base and cap sets are `base` and `cap` respectively, while `space` returns their union – all of the points in a sieve.

Returning to the question of completion, we note that during cone completion no new base points are introduced on any given process. Only additional cap points, present in the remote cone over an existing base point, are added together with the corresponding arrows. Similarly, during support completion no new cap points are introduced, so that recursive invocation of the completion methods is necessary for the total completion of the local sieves. Since in most application this is rarely required, and due to the complexities of communicating the nodes *and* arrows, we do not implement this procedure internally.

Sometimes it is useful to expand the definition of the base and cap sets by intro-

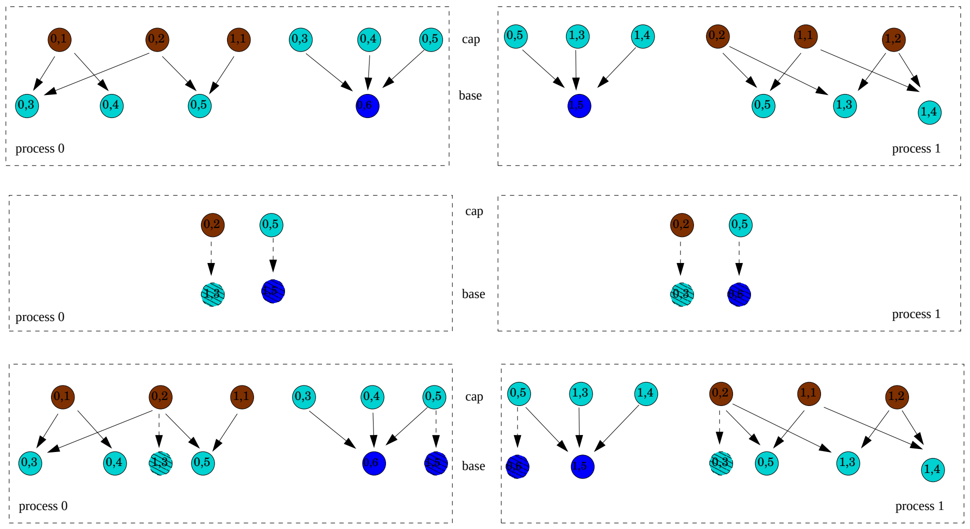

Fig. 4. A base-cap view of the mesh sieve (top), its completion (middle) and the result of addition of the completion (bottom).

ducing into them the nodes with no arrows at all. In this case base and cap can be viewed as the sets from which the terminating or originating arrow nodes can be drawn. These "placeholder" nodes can be effectively used for distributing data from a subset of processes to the rest of the communicator universe. For example, a local sieve can be prepared on the root process assigning mesh elements residing in the cap to the nodes representing the processes residing in the base. Each of the remaining processors add a single empty node to the base representing the process itself, and upon cone completion each process contains exactly the mesh elements assigned to it in the cap of the completion sieve. An example of this process can be found in Section 7.3.

The origin, i.e., the contributing process, of the nodes added during completion is optionally recorded in a *footprint* sieve with arrows from the nodes to the originating processes, themselves encoded as sieve nodes. This illustrates the utility of sieves even for sieve-specific bookkeeping purposes. In fact, the cone and support completion operations can be implemented by a single generic completion routine. Not only is a single routine easier to maintain and optimize, but this generality makes parallel, dimension independent code possible. More generally, we believe that Sieve and its completion methods can encode a very wide variety of unstructured distributed data and associated communication procedures used in scientific computing and beyond. The scalability of the communication methods is limited at most by memory requirements of order P, the size of the communication universe, or the communicator [MPI Forum 2005]. With the advent of architectures with hundreds of thousands of processors, such as IBM's BlueGene/L [Fitch et. al. 2005], this may become a limiting factor and some sort of hierarchical approach to communication may be warranted. Such organization is also likely to be reflected in the underlying architecture of the future massively-parallel computers, as is evident from BlueGene/L's experience.

2.1.3 *General Manipulation.* Finally, we list some of the basic sieve operations. All main sieve construction operations are defined in terms of arrows between cap and base points. A point p can be added to the base or the cap set, unless it is already there, by `addBasePoint(p)` or `addCapPoint(p)`, or to both sets by `addPoint(p)`, with its cone and support sets initially empty. An arrow between a cap point q and base point p is added by `addArrow(q,p)`, which will insert the appropriate points into the cap and base sets. An arrow is removed by `removeArrow(q,p)`, while leaving the endpoints in the cap and base sets respectively, even if the support of q or the cone of p becomes empty upon the removal. To get rid of the points themselves, along with all of the arrows to or from them, use `removeBasePoint(p)` or `removeCapPoint(q)`, or `removePoint(p)` to do both at once. These "precise" manipulations are useful for "sieve surgery" as exemplified in Section 7.1).

In addition to these basic operations, wholesale addition of a chain `c` to the cone or support of a point `p` is accomplished by `addCone(c,p)` or `addSupport(q,c)`, which will add all of the arrows from the points of `c` to p or from q to the points of `c`. Likewise, `setCone(c,p)` and `setSupport(q,c)` will *replace* the existing arrows. Mass point removals, such as `restrictBase(c)` or `restrictCap(c)` retain only the nodes from chain `c` in the base or the cap of the sieve, eliminating the others along with the arrows to or from them. For example, these can be used to eliminate the spurious nodes from the mesh partition sieve on the root process as in the example of Section 7.3.

Since sieves are acyclic graphs, each node can be ascribed a *height*, measured as longest path from any leaf node, and a *depth*, measured as the longest path from any root node, while the sieve as a whole can be given a *diameter*, measured as the longest path between any root and any leaf. These quantities are retrieved with `depth(p)`, `height(p)`, and `diameter` respectively. The sieve can be viewed as being divided into *strata* (sets) of points at different heights from 0 to `diameter`, or into strata of points at different depths from 0 to `diameter`. For a mesh sieve, such as that in Figure 2, the points at depth 0 are the mesh vertices, those at depth 1 are the edges, and so on. In Figure 2 the strata are indicated by color. In general, in a mesh sieve the points at depth k are k-dimensional cells, and the points at height k are k-codimensional cells, with the mesh dimension equal to `diameter`. The chains containing these strata are retrieved by `heightStratum(k)` and `depthStratum(k)` respectively.

2.1.4 *Stacks.* In many applications, it is useful to distinguish between different "kinds" of arrows. For example, in Section 7.5 we use a covering of mesh elements for both topology and assignment of degrees of freedom to elements. To accomplish this, we provide a composition mechanism embodied in the *Stack* interface, which links two sieves by using *vertical* arrows. The two sieves function as the base and the cap of the stack and can be shared among multiple stacks.

Stack is a subclass of Sieve and therefore inherits all the sieve operations; however, they now operate over the vertical arrows. New methods include `setCap` and `setBase`. The only controvesial point concerns stack point insertion and deletion, since these translate into nontrivial structural changes in the cap or base sieves. To prevent unintended side effects we block point insertions and deletions, including those implied by `restrictBase` and `restrictCap`; these methods remove arrows only in the context of stacks. As arrows can be added only between existing cap and base points, the necessary insertions must be performed explicitly on the corresponding sieves returned by the overloaded `cap` and `base` methods.

3. DATA STORAGE ABSTRACTIONS

Sieved arrays are envisioned as a discrete counterpart to the continuum concept of a field. Intuitively, fields represent quantities with spatial extent defined over some domain. They can be restricted to convenient subdomains or even defined locally, manipulated locally and then (re)assembled if they agree on the intersections of the subdomains. To represent fields in the discrete setting we choose to formalize this "sewing together" property in terms of the covering relation on the sieve of subdomains. If we reverse the covering arrows and replace subdomains with fields defined over them, we obtain a sieve with the arrows signifying restriction relations that meet some natural consistency constraints. These restriction relations between fields are the dual of the covering relations of the sumdomains, and the core of finite element methods.

The *SievedArray* abstraction formalizes the notion of a discrete field as a container of (distributed) numerical data that can be addressed at different levels of granularity reflecting the organization of the underlying "base" sieve[3]. The base is regarded as a representation of some computational domain, typically a mesh, and its decomposition into progressively finer covering subdomains. A chain of base sieve nodes is selected as the "support" of the sieved array: $\mathtt{supp}(X)$. A sieved array X is indexed by its support nodes: we can retrieve the array values "residing" at a given support node p into a contiguous "native" array $X(p)$ using the $\mathtt{getValues(}p\mathtt{)}$ method. This retrieves the data stored "at p" in the same way as the value at a given index i of an ordinary array is returned in a single variable. Similarly, $\mathtt{setValues(}p\mathtt{,values)}$ sets the new values of $X(p)$ supplyed in a contiguous native array `values`. The only guarantee made at this point is that the values are retrieved in the same order as they are set.

Any "subarray" $X(p)$ can be *refined* by *restricting* it to each node q of the cone over p thereby generating a subarray $X(q)$ at each q. This way each covering arrow $q \rightarrow p$ is converted to an oppositely oriented restriction arrow $X(q) \leftarrow X(p)$. After such a refinement p is replaced by its cone in the support of the sieved array and the values can be addressed at a lower level of granularity. Indeed, instead of a single array $X(p)$ we can retrieve any one of the "smaller" arrays $\{X(q)\}$. In general, to any support chain $\mathtt{c} = \{r\}$ there corresponds a *cochain* (a chain in the dual sieve) of arrays $X_{\mathtt{c}} = \{X(r) : r \in \mathtt{c}\}$. Indeed, to each covering relation between the elements of chains there corresponds a dual restriction relation between the corresponding cochain elements. Likewise, for any complete covering $\mathtt{c}' \twoheadrightarrow \mathtt{c}$ the corresponding cochains are in a *refinement* relation: $X_{\mathtt{c}'} \twoheadleftarrow X_{\mathtt{c}}$; in the discussion of a single cochain element refinement we have $\mathtt{c} = \{p\}$, $\mathtt{c}' = \mathtt{cone}(p)$, and $X_{\{p\}} \twoheadleftarrow X_{\mathtt{cone}(p)}$.

[3]As we shall see below, the subdomain sieve is typically represented as the base of an appropriate stack.

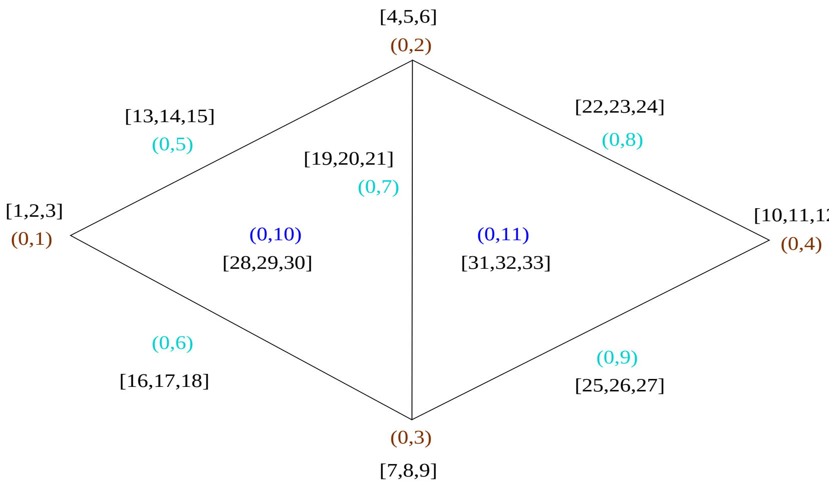

Fig. 5. Illustration of a discrete field defined over the doublet mesh. Each element carries three numbered degrees of freedom indicated in square brackets. A sieved array can be defined by associating with each mesh sieve element the degrees of freedom contained in its geometric closure.

Thus, a sieved array is a collection of cochains that are refinements of one another. At any given time a single cochain represents the sieved array. The following example illustrates the introduced concepts and introduces the bundle construction used in the implementation. Consider a discrete field defined on a doublet mesh by attaching 3 degrees of freedom to the interior of each mesh element (Figure 3). Such a field may result, for example, from a mixed order finite element discretization of some continuous fields over the domain discretized by the mesh.

The assignment of degrees of freedom to mesh elements can be encoded by the vertical arrows of a stack, which we call the *bundle*, whose cap contains the degrees of freedom themselves in a discrete[4] sieve, and the base is the mesh sieve, sometimes referred to as the *topology*. The bundle is set and retrieved using `getBundle()` and `setBundle(bundle)` respectively, where any object implementing the Stack interface can serve as the `bundle`.

Given a base element e, the vertical cone over its *horizontal closure* represents the degrees of freedom *supported* at e (see top of Figure 3). If e is among the support elements of the array, then using e as an index into the array will return only the values of the degrees of freedom supported on e in a contiguous native array.

A sieved array starts out supported on the leaves of the base sieve (chain $\mathtt{c}_0$) and in the course of computation may be refined to have a more convenient support chain `c`. The `refine(c)` method will refine the each of the element of $\mathtt{c} \subset \mathtt{supp}(X)$. The inverse operation of *assembly*, discussed below, is implemented by the method `assemble(c)`, takes the same input as `refine`. It reconstructs the subchain supported on `c` from its complete covering $\mathtt{c}' \subset \mathtt{supp}(X)$ and replaces $\mathtt{c}'$ with `c` in

[4]Discrete, in this case, means 'without arrows', by analogy with a discrete category, which such a sieve uniquely determines. This is also reminiscent of a space with discrete topology; in this case, it means 'absence of coverings'.

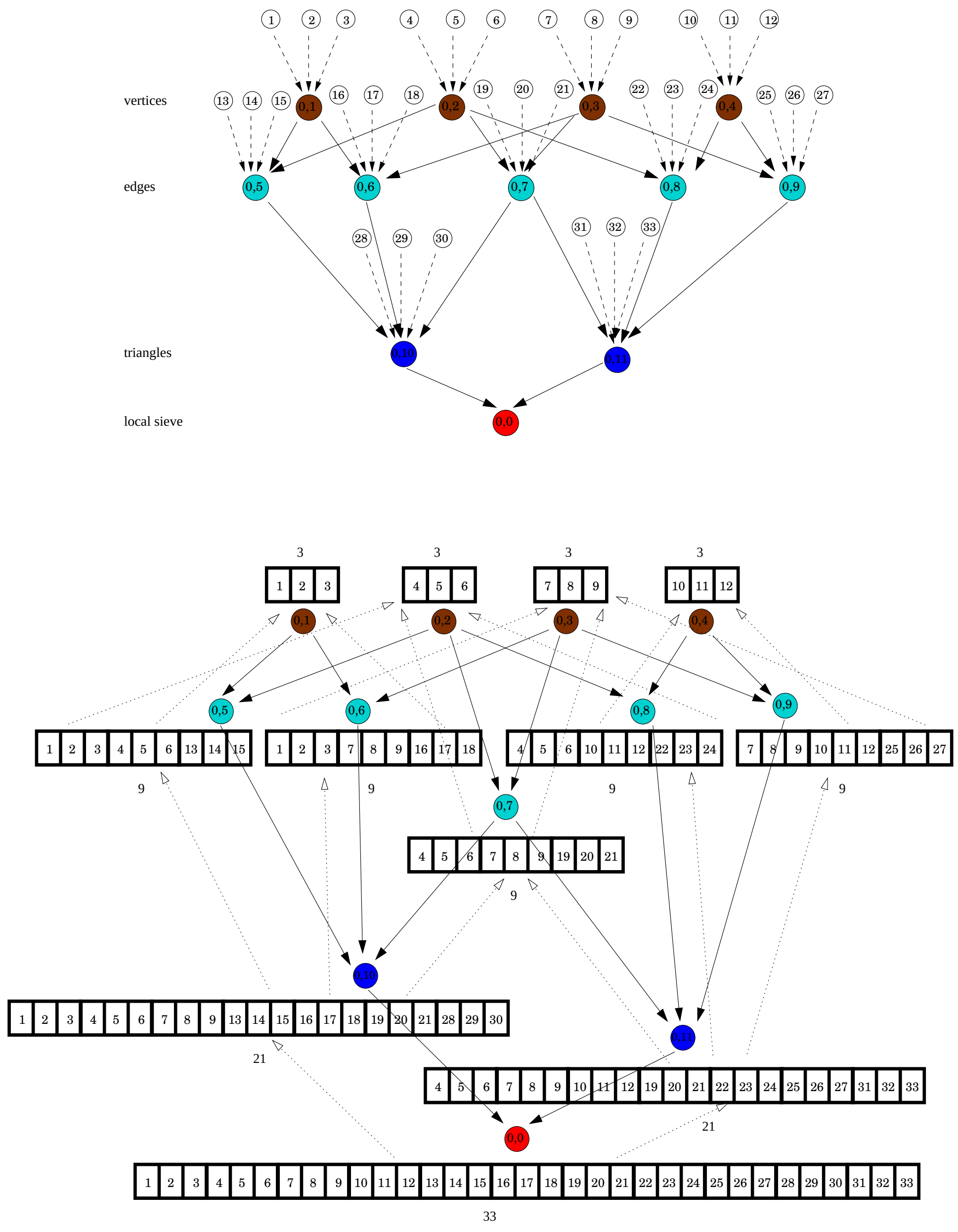

Fig. 6. Illustration of a *bundle* stack (top), assigning degrees of freedom to the elements of the doublet mesh by vertical arrows (dashed), and of the dual sieve (bottom) with each support element replaced by a contiguous array of values corresponding to the degrees of freedom over its closure. The dotted arrows of the dual sieve represent restriction relations.

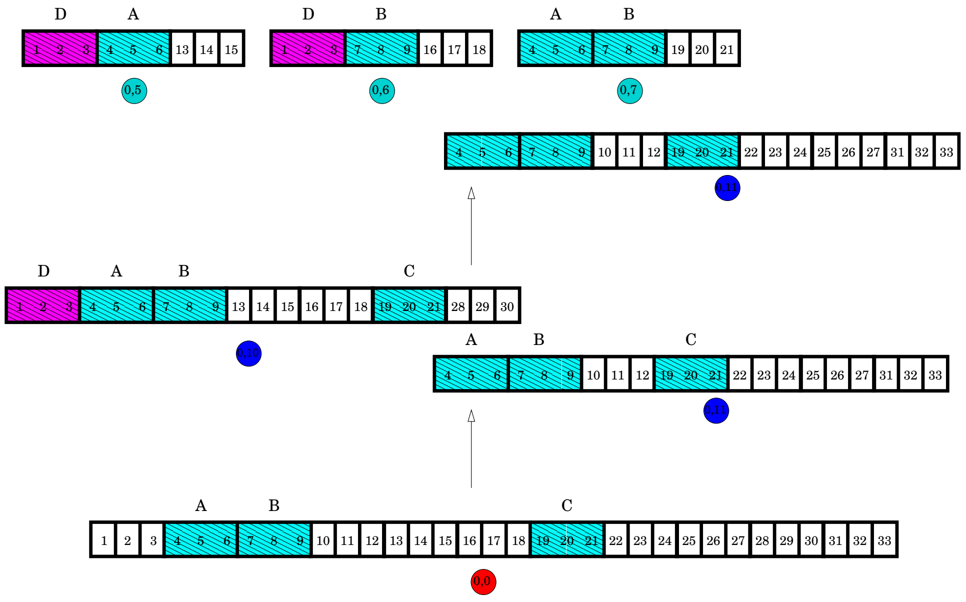

Fig. 7. Illustration of a refinement process on a sieved array over the doublet mesh sieve. From the bottom up: first the single node leaf support chain (`totalSieve`) is refined into its two covering nodes (mesh triangles), then the left triangle is refined into the set of covering edges. The degrees of freedom duplicated at each state are indicated by color.

$\mathtt{supp}(X)$. Another method, `restrict(c)`, leapfrogs all of the intermediate levels of refinement between $\mathtt{supp}(X)$ and $\mathtt{c}$, where $\mathtt{c} \twoheadleftarrow \ldots \twoheadleftarrow \mathtt{supp}(X)$, and refines X all the way to $\mathtt{c}$. Assembly can be done using $\mathtt{assemble}(\mathtt{c}')$ where $\mathtt{c} \twoheadrightarrow \mathtt{c}'$.

Since the leaves become roots in the dual sieve, the initial state is a cochain of all roots, or *root cochain* $X_{\mathtt{c}_0}$. In some situations it may be convenient to adjoin a unique leaf node to the base sieve or stack signifying the total (or local) sieve, as illustrated in Figure 3. After the maximal number of refinements the support reaches the roots $\mathtt{c}_\infty$ through a series of complete coverings $\mathtt{c}_\infty \twoheadrightarrow \ldots \twoheadrightarrow \mathtt{c}_0$, and the corresponding cochain is reached from the coroot chain through a series of refinements $X_{\mathtt{c}_\infty} \twoheadleftarrow, \ldots \twoheadleftarrow X_{\mathtt{c}_0}$, as illustrated in Figure 3.

More generally, it may be desirable to have each cochain element support represent a chain of mesh elements (rather than a single element) that are not readily present in the mesh sieve. For an example think of a decomposition of a mesh into the boundary and the interior, the set of submeshes employed in a domain-decomposition method, or progressively finer meshes of a multilevel method. Each chain of elements can be represented by a single stack point covered by the corresponding mesh elements, residing in the cap of the stack. This additional structure can be conveniently represented by setting this new stack into the base of the bundle (recall that a stack is a sieve by inheritance), as illustrated in Figure 8. The degrees of freedom at any support point are easily calculated as a composition of cone operations: the result of the cone operation in the inner stack is used in the input for the cone operation in the outer stack. To disambiguate the situation we

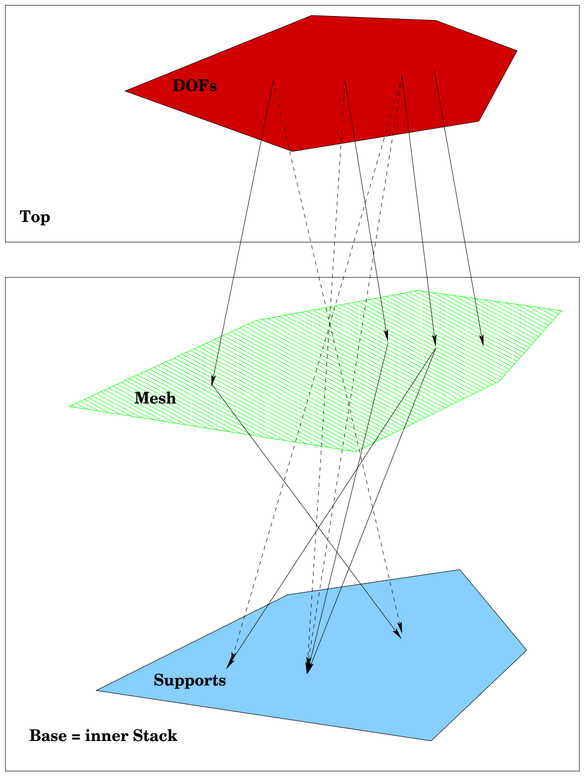

Fig. 8. Illustration of a nested stack bundle construction: the internal base stack contains supports representing mesh element chains. Dashed arrows are the result of composition of the arrows from two stacks – inner and outer (bundle) – and can be stored explicitly as a form of caching.

will refer to a *base stack* to denote the base of the bundle, and a *base sieve* to denote the top of the base stack. If the base stack is in fact no more than a sieve, the two notions coincide.

3.1 Refinement

It appears clear from the example above that the values of different cochain elements obtained by refinement of a root chain need not be independent. In fact, these dependencies are encoded in the covering structure of the base sieve via the restriction procedure. The nodes covering p presumably all obtained their values from a single source, a cochain element at p, so we expect correlations between them. This notion can be made precise if we consider iterated restrictions. Consider the following diagram

$$\begin{tikzcd}
& r \arrow[dl] \arrow[dr] \arrow[dd, dashed] & \\
q_1 \arrow[dr] & & q_2 \arrow[dl] \\
& p &
\end{tikzcd} \tag{1}$$

The dashed vertical arrow denotes the composition of the diagonal sieve arrows and is not necesserily present in the sieve. Still, we demand that there be a unique restriction $X(p) \to X(r)$. Naturally, there are two candidates for the definition of this restriction – the iterated restrictions $X(p) \to X(q_1) \to X(r)$ and $X(p) \to X(q_2) \to X(r)$ obtained by following the two paths from r to p composed of the explicitly stored sieve arrows. We now impose the requirement that the two compositions be identical thereby *defining* the unique restriction from p to r. It follows that a restriction operator $\mathrm{res}_p^{p''}$ is defined for any pair of points connected by a path of zero or more arrows $p'' \to \cdots \to p$ satisfying the composition requirement[5]

$$p'' \to p' \to p \qquad \Rightarrow \qquad \mathrm{res}_{p''}^{p} X_{\{p\}} = \mathrm{res}_{p''}^{p'} \circ \mathrm{res}_{p'}^{p} X_{\{p\}}. \tag{2}$$

In fact, in the example above, the proviso regarding the taking of horizontal closures in order to determine the degrees of freedom residing at each cochain element ensures that the restriction operator `res` respects the composition of covering arrows in the base sieve, as expressed in Equation 2.

Returning to the Diagram 1, since r lies in the meet of q_1 and q_2, the just introduced consistency requirement states that the cochain elements $X(q_1)$ and $X(q_2)$ *agree* on the intersection in the sense that the cochain generated on the points of the meet of q_1 and q_2 via refinement of $X_{\{q_1\}}$ or $X_{\{q_2\}}$ are independent of the source and the path refinement takes. A cochain such as $X_{\{q_1,q_2\}}$ whose elements agree on their meets is called a *cocycle*. In the simplest case, such as the example in Figure 3, the related values are duplicated at each of the nodes with nonempty intersections.

From the property of the restriction operator (2), it follows that cocycles are preserved under refinements. Clearly, a cochain defined on the root chain of a sieve is a cocycle, since roots have empty cones, and hence empty meets. If the base sieve possesses a single leaf, such as in Figure 3, any cochain defined on it is a cocycle, since clearly any cochain element agrees with itself. In fact, any cochain obtained from such a singleton root cochain will be a cocycle.

[5]This gives the assignment of cochains to sieve nodes, with the restriction operators as described, a functorial character.

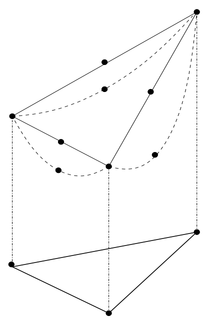

Fig. 9. Illustration of *distinct* P_2 discretizations of scalar fields represented by sieved arrays over a triangle boundary with *identical* refinements down to the vertex level.

3.2 Assembly

A fundamental feature of continuum[6] fields is the *equivalence* of local and global descriptions given by cocycles. Clearly, a global field over some open domain $D = \bigcup D_i$ defines a cochain of fields with analogous properties (continous, smooth) by restriction to a chain of open subdomains D_i constituting a covering of D. Conversely, a cochain of local fields *uniquely define* a global field precisely because they agree on overlaps, so that *any* cocycle is a refinement of a global field.

This characterizes continuous fields as a *sheaf* over the underlying domain, which can be concisely rephrased as "a field is fully determined by its restrictions". It is a property of the restriction operator that is rooted both in the nature of the fields, which are defined by their pointwise values, and the nature of the covering – a set-theoretic covering by open sets, – which ensures that each point is represented in at least one subdomain.

If the analogous sheaf property is to hold in the discrete case, a sieved array must be uniquely defined by its refinement to the root chain of the base sieve. However, this is not the case for the example in Figure 3: the cochain values at the finest level reflect only 12 degrees of freedom, while the whole array is specified by 33! Similarly, a piecewise quadratic representation of a continuous scalar field defined at the boundary of a triangle can be represented by a sieved array with supports at the nodes indicated in Figure 9. However, its refinement to the vertex level can be obtain by restriction of a whole family of piecewise quadratic fields, since the interior edge node is omited from the vertex-based description.

What went wrong? The main problem is a flawed geometric interpretation of a purely syntactical construction represented by the mesh sieve. Indeed, when attaching the degrees of freedom we identify the sieve nodes with mesh elements' *interiors*, while when retrieving the cochain values the same nodes are identified with the *closures* of the corresponding elements, and their coverings are identified with the *boundary* during refinement. In short, the topology of the mesh sieve is inadequate for a consistent defintion of *this* discrete field.

The situation can be rectified in various ways, but essentially all of them amount

[6]The adjective "continuum" in this context is used as the opposite of "discrete".

to supplying a better topology for the base of the sieved array. The basic structure of coverings among mesh elements is sufficient to express only piecewise linear discretizations of continuum fields adequately, and mostly likely on simplicial meshes only. Higher order methods typically attach degrees of freedom to element interiors (except Hermite-based, perhaps), requiring a finer topology and a richer sieve, as in the example above and Figure 9.

The simplest solution would be to cover each (nonroot) element e with a new node e° representing the interior of e. A refined cochain would now be supported on the boundary *and* the interior. Such an augmentation, however, would nearly double the size of the base sieve while addiing essentially no new information about the structure of the space – an interior node has exactly one arrow attached to it and everything about it can be inferred from the element node.

3.3 Sifting

We can shift some of the complexity from the *structure* of the bundle to the *functionality* of the interface controlling access to the array data, which we will term a *Sifter*. A Sifter, whose function may be directly implemented by a sieved array class, implements a particular *sifting policy*. The policy defines, among other things, the inputs acceptable as indices into a sieved array, even though they may not be present in the base stack. For example, assuming only positive `(p,i)` pairs are used to label the base sieve nodes, a node labled by `(-p,-i)` may be recongnized, within a given sifting policy, as the *interior* of a valid base node `(p,i)`. A cochain supported on `(p,i)` is then implicitly supported on `(-p,-i)` and the corresponding cochain element contains the interior values only. Likewise, a sifter may admit chains of base stack points as input to encode a finer indexing of cochain data.

The main functions of a sifter, and the sifting policy it defines, is to define the effect that the refinement (and restriction, on which refinement depends) and assembly operators have on the array data. In the simplest, yet most ubiquitous case, as in Figure 3, restriction amounts to reordering and gathering of data. More broadly, restriction may involve interpolation or more general coordinate transformations on the data, when transfering it from one level of granularity to another. For example, the underlying continuum field may take values in a topologically nontrivial manifold, such as a sphere. The values of the corresponding global discrete field can be stored in a sieved array in an arbitrary fashion. Upon refinement to the level when each elements maps into a single coordinate system, restriction will perform the appropriate coordinate transformations. The exact input sequences indicating the coordinate system will be dictated by the sifting policy of the sieved array.

A particularly simple, yet important, example of a coordinate transformation is array component reordering. This may be necessary, among other situations, if the user expects the output of cochain subscripting to be stored in some desired order. For example, in using higher order finite element methods, say P_2, retrieval of the discrete field components supported at a mesh edge must reflect the interpolation node order. There are 3 nodes per edge in this case: one per each of the two endpoints e^0_1, e^0_2 and on in the edge itself (interior) e^1_1. The two orientations of the edge can exactly specify the two corresponding order of retrieved components. An

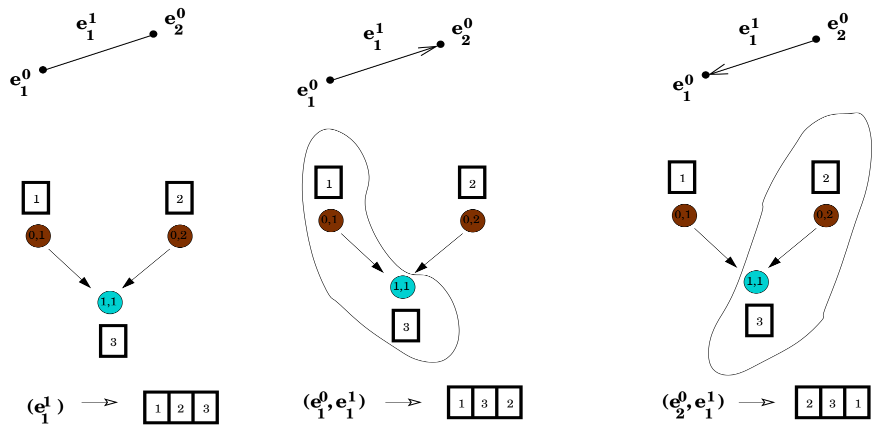

Fig. 10. A coordinate transformation of the cochain subsripting output consisting in reordering. Indexing by the edge element outputs the data supported on its closure in an arbitrary ordering (left). Indexing by cell-tuples forces a prescribed ordering (middle) and (right).

orientation can be unambiguously prescribed by a *cell-tuple* [Brisson 1989] consisting of an endpoint and its supporting edge. Admitting element chains representing such cell-tuples into the bundle structure (representing them by base points in the bundle), or into the sifter interface (allowing as input) provides the user with the capability to specify the desired output order, as illustrated in Figure 10.

In general, use of cell-tuples corresponding to paths through the mesh sieve allows for a unique ordering of the faces of the mesh element at which the path terminates [Brisson 1989], and hence for a unqiue ordering of the cochain data indexed by that element. The idea behind this relies on the use of generalized barycentric subdivisions for the specification of a unique coordinate system on each cell of a complex. It is particularly transparent in the simplicial complex case: the origin of the coordinate system specified by the initial vertex of the chain tuple, the first coordinate direction from that point is along the edge following the vertex down the chain. The second coordinate direction is any direction complementary to the edge in the the triangle – the next element in the cell-tuple after the edge – containing the edge, and so on. The simplest case of a two-dimensional simplicial complex with the usual barycentric subdivision is illustrated in Figure 11.

Allowing nontrivial transformations of the cochain data during refinement, restriction and data retrieval makes sieved arrays suitable for the modeling of sections of *fiber bundles* over base spaces with discretizations encoded by sieves in terms of coverings. In fact, sieved arrays can be interpreted as local sections of fiber bundles over the underlying sieves. In all of these situation, simple and complex, the only common constraint placed on the restriction operator is expressed by Equation 2.

Assembly, on the other hand, is inextricably linked to restriction because, among other things, it is the process of recovery of a coarser cochain from its refinement. We may insist that both refinement and assembly be reversible and act as each other's inverses. As we have seen (e.g., in Figure 9 and Figure 3), this is impossible in general, since the assembly operator is not epimorphic when acting on the values

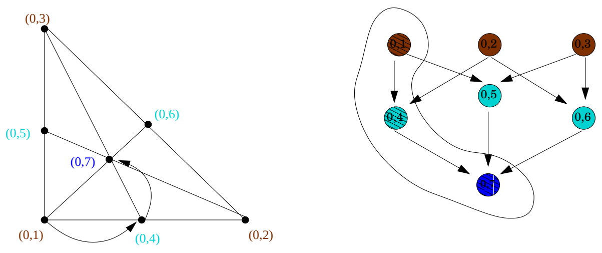

Fig. 11. An orientation on a simplex defined by an ordering of the vertices of a barycentric subdivision, or, equivalently, by a path from a root node (0,1) to a to a leaf node (0,7) (enclosed, nodes hatched).

of the fine cochain alone. We can weaken the inversion requirement to demand that refinement is a left inverse of assembly. In other words, given a support chain `c` and its complete covering by `c'`, `c'` $\twoheadrightarrow$ `c`, if $R^c_{c'}$ is a refinement operator and $A^{c'}_c$ is the corresponding assembly operator, then

$$R^{c'}_c \circ A^{c'}_c = I_{c'}. \tag{3}$$

where $I_{c'}$ is the identity acting on the cochains supported on c'.

In practice, we do not expect refinement to discard data but simply to expose the appropriate cochain representing the array at a certain level of granularity. Therefore, since assembly acts on all of the sieved array, not only on the exposed cochain, the ambiguity in the reconstruction of a coarser cochain array can be removed, and the other inversion identity will hold:

$$A^{c'}_c \circ R^c_{c'} = I_c \tag{4}$$

This way assembly can be interpreted as an *update* of the sieved array from the current representative cochain. Refinement can then be viewed as splitting a coarse cochain into complementary parts, one of which, the refined cochain, is open for query and updates, while the data over other is kept fixed. Then assembly of the coarser cochain consists of reconstructing it from the two pieces, the updated and the fixed.

In this light it appears particularly natural that the life cycle of a sieved array should start at the coarsest level. In practice, however, a sieved array is referenced at the same level of granularity throughout the computation[7]: it is refined to the appropriate level and cochain elements indexed by the suitable elements of the refined support are examined or assigned. Assembly is performed occasionally to enforce the cocycle condition, and thus to produce a global field over the `totalSieve`, followed by refinement back to the "working level" of granularity. To encapsulate this commonly occuring sequence of operations we provide an equivalent `update(c)` method, where assembly is done down to the level of chain `c`. An empty chain is conventionally used to denote the `totalSieve`, and hence the "total assembly".

[7]A notable exception is furnished by multilevel methods.

4. OPERATORS

The role of assembly is not merely reconstruction of coarse cochains from the covering cocycles. A more important function is the assembly of cochains that *are not* cocycles (i.e., whose elements do not agree on the meets of the supports). This situation arises commonly in application of operators to discrete fields. In a typical continuum setting, a differential operator acts equally well on the global and local fields generating the same pointwise values on output, which can then be trivially assembled for any cocycle. Indeed, being local, a differential operator requires only the data from an arbitrarily small neighborhood of a point, which is available in full for any local field over an open subdomain.

4.1 Finite Difference sifting

In a typical discrete setting, the situation is markedly different. A discrete operator P arising as a discretization of a differential operator acting on a local portion of a discrete field X defined over a subdomain of a mesh, can generate only partial data for the resulting discrete field Y. If finite-difference discretizations are used, the operator can only compute values of Y at "interior" points – a designation depending on the stencil being used – with the rest computed on different subdomains and designated as "ghosts". The refinement procedure in this case consists of *replacement*, where a single process (e.g., with the lowest rank) containing a given mesh element e in the interior of a locally structured grid block scatters its local cochain element $X(e)$ to all other processes storing the same data.

4.2 Finite Element sifting

Finite element methods are more symmetric in the sense that operators discretized this way typically contribute output data for any input degree of freedom. However, the input data and the output contribution correspond only to the components of X and Y along the local basis elements supported on the local subdomain. The output contributions must be incorporated (e.g., added) from each subdomain to obtain a complete representation of the result in the finite element basis. When representing this situation in terms of sieved arrays, the refinement-assembly pair of operators act as a *partition of unity*, resolving a field, usually using projections, into components supported at a given covering of the domain by subdomains, and then assembling results, usually by addition, in general using a linear map.

This discussion demonstrates that assembly must be general enough to assemble cochains that are not cocycles, while on cocycles it must reproduce the expected results (i.e., the array whose restrictions generate the cocyle). This property can be nicely expressed using the update operator introduced above. Denoting the support of a sieved array by c', the update operator by $U_{\mathsf{c}}^{\mathsf{c}'} = R_{\mathsf{c}}^{\mathsf{c}'} \circ A_{\mathsf{c}'}^{\mathsf{c}}$ and comparing with (3) we conclude that $U_{\mathsf{c}}^{\mathsf{c}'}$ reduces to the identity operator on c'-cocycles, while all other chains are "projected" on this cocycle space.

4.3 General linear sifting

We can admit into refinement-assembly pairs the restriction-prolongation operators of multilevel methods. Here different supports of a sieved array may correspond to meshes of different resolution with covering arrows encoding the relations between

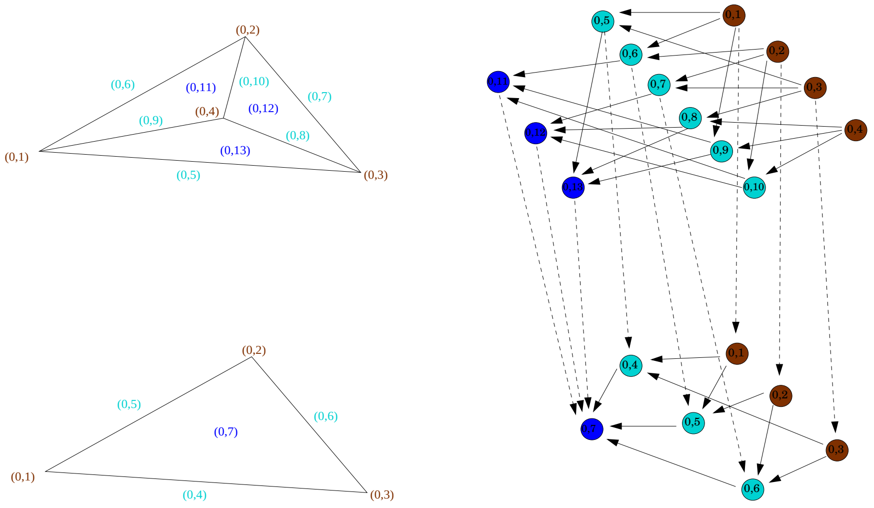

Fig. 12. ...

the elements at different levels. This can be conveniently represented using stacks with the base and the cap being sieves representing meshes at different levels (see Figure 12). A sieved array defined over such as stack would represent a field at different levels of resolution at once. Clearly, in this case we cannot insist on a unique reconstruction of a fine state from a coarsened state, although as mentioned before, if assembly is viewed as an update, it is possible in practice. Similarly we can deal with the assembly of sieved arrays defined over nonconforming meshes as in Figure 13. In both of the above cases the construction of the refinement and assembly operators involves nontrivial choices, such as the interpolation or averaging methods, which are highly problem dependent.

Together the refinement and assembly operations define the *sifting policy* implemented by the sifter, which can be thought of as representing a particular class or space of sieved arrays with operators acting on the appropriate spaces only. Operators themselves can be easily represented by sieved arrays. Indeed, if the local action of an operator, application to a chain at a given support node generating the values of a chain at the same node, depends on some numerical values, such as local matrix coefficients (Jacobi matrix in the FEM case), they can be stored in another chain with the same support. The application of an operator $X \xrightarrow{P} Y$ then consists in refinement of a sieved arrays representing X, Y and P to the necessary level, local calculations of Y values followed by assembly, if necessary, as illustrated in Figure 4.3.

The assembly of a fully distributed field proceeds as an assembly from any covering. The `totalSieve` can be viewed as a sieve node covered by the `localSieve` support points residing on different processes with the same sifting policy applied on the meets of local sieves when assembling the chain supported at the total sieve node, the coarsest array state.

Figure 15 illustrates different assembly procedures. At the top of the figure a

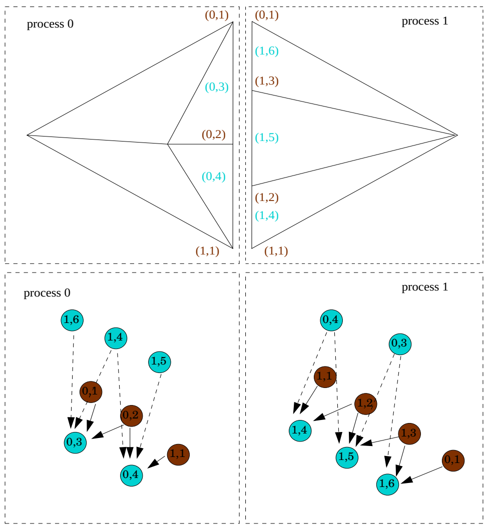

Fig. 13. ...

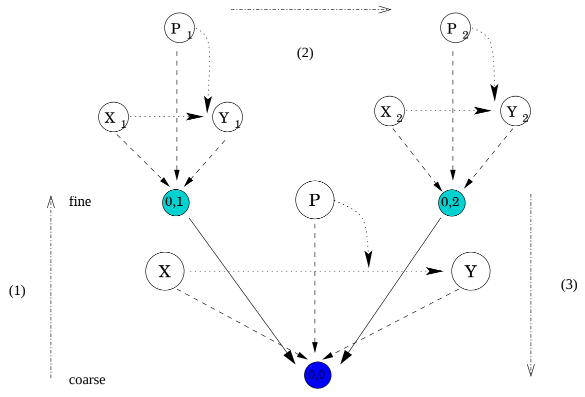

Fig. 14. Illustration of an action of a sieved operator P on a sieved array X producing a sieved array Y, all sharing the same base sieve. The refinement-assembly resolution of identity intervenes at stage 1 (refinement), and at stage 3 (assembly) after the local operator application generates a refined Y in stage 2.

cochain of two matching scalar fields discretized using piecewise-linear elements. If those values disagree, they must be coerced to produce a consistent field, which is illustrated at the bottom of Figure 15.

If we think in terms of the old contiguous vector approach to FEM, we can liken the approach embodied in sieved arrays to indexing into the storage using mesh elements rather than integers. In fact, the common domain decomposition approach to parallelism, used in the Portable Extensible Toolkit for Scientific computing (PETSc), can be seen as a special case of refinement. The field is refined to a collection of subdomains, serial calculations are done, and the results are combined

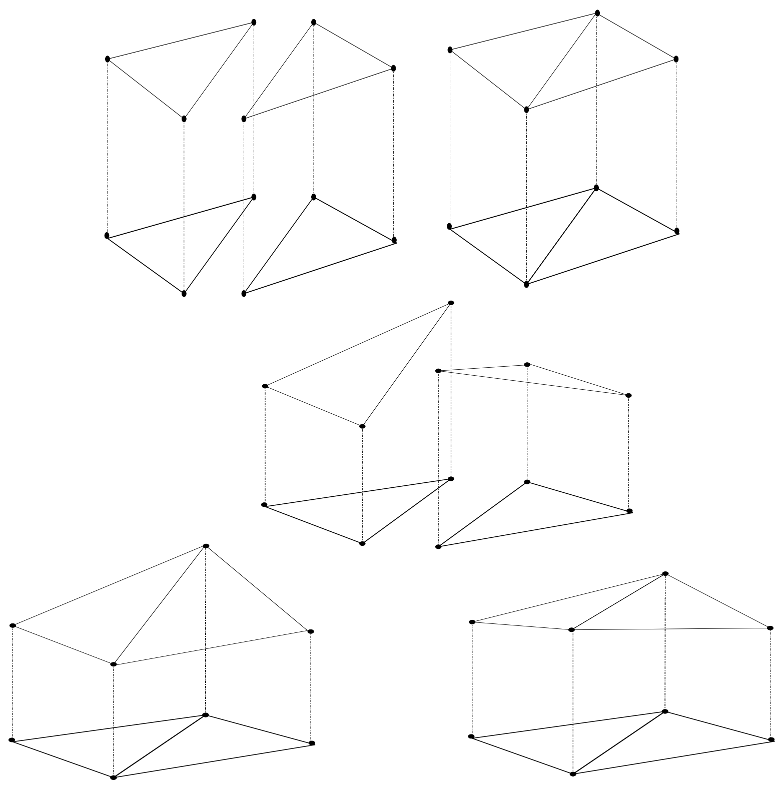

Fig. 15. Illustration of assembly of a sieved array over a doublet mesh. A cochain over the triangles matching on the intersection (top), and not matching (middle). In order to define a field on the total mesh, the non-matching cochain must be *coerced* by one cochaing taking precedence (bottom left), or averaging (bottom right).

along the parallel interface.

5. MESH

Now we have defined all of the ingredients of a flexible mesh representation. The topology of a mesh is represented by a sieve. The boundary submesh can be implemented as a stack with the bottom containing the subsieve representing the boundary with the induced covering relations, the top being the mesh itself, and

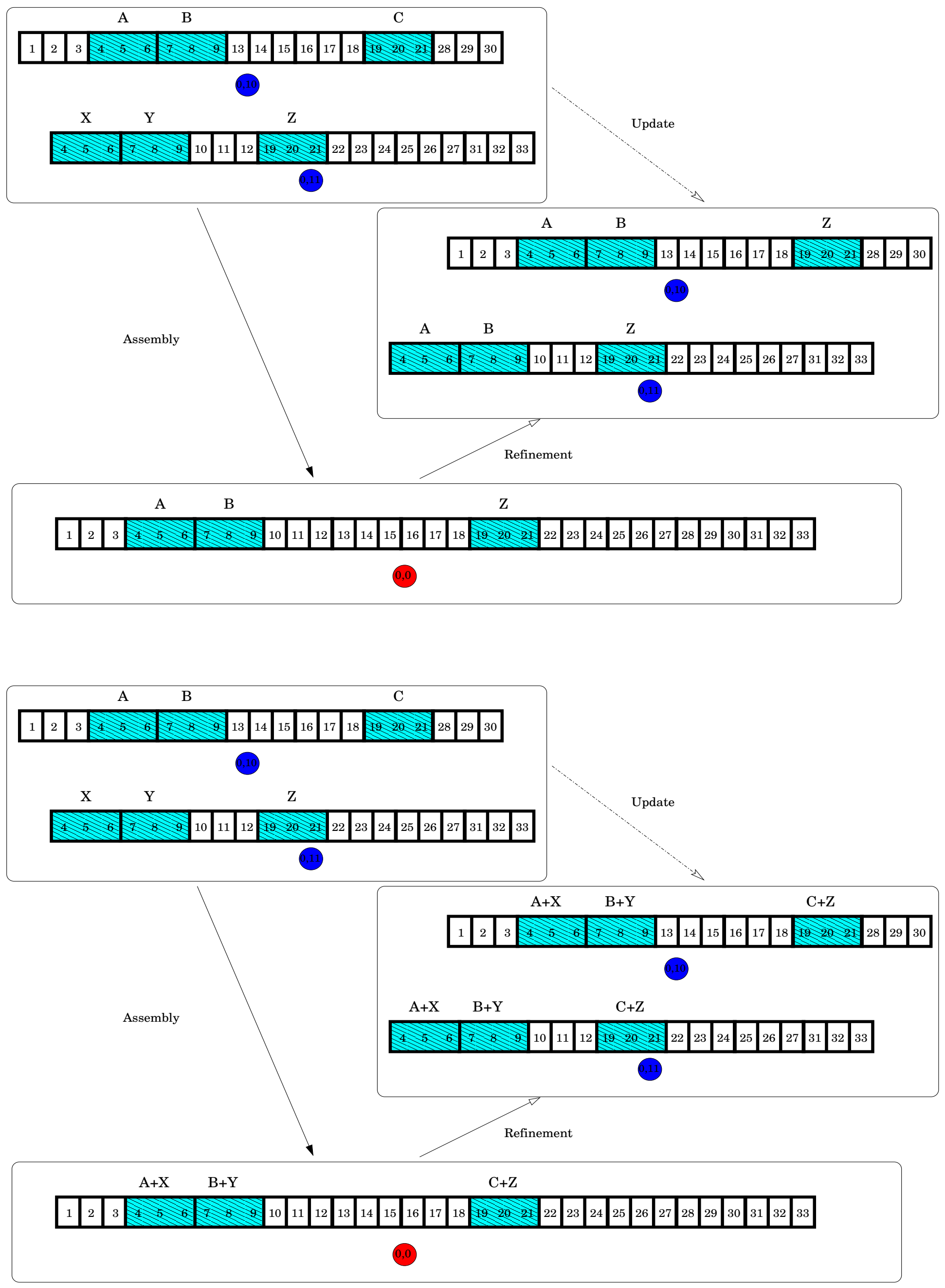

Fig. 16. Illustration of array assembly and update implementing the replacement policy (top) and addition policy (bottom).

the vertical arrows identifying the elements of the boundary with their embedding into the full mesh. The boundary elements in the full mesh are easily identified as the vertical support of the total boundary sieve.

Analogously we can represent any subsets of a mesh or its partition into subdomains, which makes discussion of field restrictions to those subdomains particularly simple, using the SievedArray interface. We feel this is superior to the representation of boundaries and subdomains using markers, since it eliminates having to sift through the element space by hand, identifying the boundary or subdomain elements by marker examination; using sieves it is accomplished by a single `cone` or `support` operation, which can be efficiently implemented, both in serial and in parallel.

As mentioned in the Introduction, it is common to include geometry in the definition of the mesh itself. For instance, Triangle [Shewchuk 2005] assumes that vertices alone carry geometric information. Other mesh formats allow coordinates to be associated with other parts of the mesh, but tend to store this information as part of the mesh data structure. This seems counterintuitive since a given topology can be embedded in many different spaces and in many different ways into the same space. Thus the geometry appears as an external property, imposed on the mesh from outside, and subject to change. It is therefore natural to separate the geometric information of the embedding from the topological information of the mesh itself.

In fact, the geometry of the mesh can simply be implemented as a field represented by a sieved array. If the embedding is done into a nontrivial space (e.g., a sphere or a torus to emulate periodic geometry), coordinate transformations that were discussed above may be necessary. In fact, the fields may take values in essentially arbitrary target spaces amenable to computational representation (think of fiber bundles with specific typical fibers). Thus it may represent tensor fields (representation of operators in Section 4 is an example of this), discrete markers and even fields with values in object classes.

The mesh generator, refiner and coarsener may now be merged into a single interface acting on a stack representing the embedding of the boundary into the mesh. A stack with an empty top indicates the request to generate the mesh from the boundary data. Its geometry can be supplied separately as a sieved array, or as a full mesh object referencing the bottom of the stack. Likewise, the refinement or coarsening constraints on the mesh can be easily implemented as sieved arrays attaching, for example, the maximal or minimal volumes to the elements to be refined.

5.1 Implementation

We provide a reference implementation of the interfaces discussed above. The core algorithms are implemented in C/C++ on top of PETSc (see [Balay et al. 2005]) and the Message Passing Interface (MPI) (see [MPI Forum 2005]). This ensures a high level of performance and scalability of the core sieve and array code.

To ensure interoperability and the ease of code deployment we employ a language interoperability environment ASE developed at Argonne National Laboratory [Knepley et al. 2005] and providing interfaces to the core capabilities in various languages. At the moment C/C++ and Python are supported, although we plan to support Fortran and Matlab interfaces in the future.

6. HOW DOES IT STACK UP?

6.1 Value of Abstraction

In short, the Sieve abstraction retains a minimum of structure to ensure the greatest possible expressivity. For example, although the connectivity, or incidence, information is not explicitly specified in a sieve-based description of mesh topology, it can be recovered through an application of a concise and elegant algorithm (see Section 7.2). Moreover, even basic topological information such as the dimension and shape of an element, though not explicitly preserved, can be recovered through simple sieve operations. However, the absence of this information in the objects and especially in the operations over a sieve make it possible to write algorithms that are independent of these extraneous details, as demonstrated throughout Section 7.

6.2 Triangle

The most common scheme for representing a computational mesh is exemplified in the format used by the Triangle [Shewchuk 2005] mesh generator. Vertices are first identified by their spatial coordinates, and then faces are specified by the collection of vertices they contain; edges are left implicit. This is also the strategy employed by the TSTT interface [Brown et al. 2005]. Here the geometric information is embedded directly into the mesh description rather than being expressed separately, as in our fibre bundle construction. In fact, material attributes are also allowed using another mechanism, when they could be handled in an identical manner using bundles. Moreover, explicit distinctions are made between topological elements of differing dimension and shape. Our sieve construction, on the other hand, treats all elements equally. This approach simplifies greatly the task of coding algorithms that are independent of the intrinsic dimension of the mesh.

6.3 Incidence Relations

We initially formulated the covering relations in terms of an equivalent *incidence* relation graded by dimension. Thus, the user could ask for all incident elements of dimension d rather than the cone or composition of cones. Restriction was similarly expressed in the interface, but its duality to the covering category was obscured and it still possessed unnecessary distinctions in dimension. The user interfaces presented here are similar; however, the basic structures are much simpler in the Sieve case and extend much more readily to other scenarios, such as parallel partitioning. Furthermore, the underlying implementation was vastly simplified by using sieves.

6.4 Ramifications and Further Applications

The Sieve concept is much more general than the traditional mesh structures commonly used for numerical solution of PDEs, and it can represent many more structures commonly implemented separately. Using sieves, we can easily define quad- and oct-tree decompositions used in fast evaluation of integral operators, such as the Fast Multipole Method. Such global computational space decompositions exploit the decay properties of integral operator kernels to agglomerate the effects of interaction of a point with a whole subdomain, if the two are sufficiently separated. Such structures map well on global physical network topologies, such as the tree network of the BlueGene/L (BGL) architecture.

Another example of nonlocal interactions can be found in metabolic networks, whose degree distribution follows a power law resulting in many highly connected nodes. Recently decomposition algorithms have been proposed for separation of such graphs into a locally connected (mesh-like) part and a global "shortcut" graph [F. Chung 2004]. Using sieves, we can model this situation and investigate different mappings of the nonlocal portions of the networks onto BGL-like communication topologies. This could open a computational avenue to problems on metabolic, or more broadly scale-free, graphs.

In general, we expect many applications of sieve-based algorithms well beyond mesh representation. We view the Sieve concept as a general tool for "geometrization" of computational problems by viewing computational objects in terms of coverings by more elementary parts. The Sifter becomes the central programmable part that must be overloaded to accomodate exotic applications; however, most users will find the basic sifting policies mentioned above sufficient.

In micromagnetics, modeled by the Landau-Lifschitz-Gilbert equations, the magnetic spin does not take values in a Euclidean vector space, but rather on the sphere. A consequence of the global topology is that a single coordinate chart cannot cover the entire space. When retrieving values in the overlap between charts, it may be necessary to perform a coordinate transformation before returning the values. This complication is reflected in our formalism by changing the *sifting* policy of the sieved array providing the values.

The computational use of fiber bundle ideas by itself raises several interesting points. For example, the computation of fields may require changes to the base sieve. Assume that for the sieved array in Figure 3 after a computational step the local field over (0,10) no longer "fits" into the given coordinate chart. Such as situation is quite possible in certain applications, such as micromagnetics. The only way to deal with this situtation may be to refine the base mesh and to generate a finer field mapping each element into a single coordinate chart.

The Finite Element Tearing and Interconnecting [Klawonn et al. 2005] method is an iterative substructuring method using Lagrange multipliers to enforce the continuity of the finite element solution across the subdomain interface. As such, degrees of freedom that lie on the interface itself are treated independently on either side of the interface, and in the calculation phase we use vectors containing both as global degrees of freedom. This can easily be accommodated by again changing the sifting policy so that on subdomain boundaries values are not reduced, but are returned independently. The interpolants do not yet match, or are cochains, preventing us from extending to a global function across both elements. However, after the solve has been accomplished, values along the interface do match, pushing the entire domain into the sheaf, and allowing us to revert to the familiar insertion sifting policy.

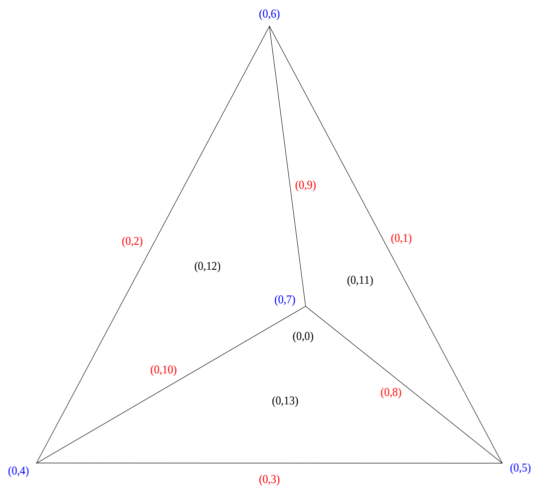

Fig. 17. Point insertion into a triangular face

7. EXAMPLES

7.1 Simple Mesh Surgery Operations

One common operation involves splitting a triangular face by point insertion. Suppose that we insert a vertex inside an existing face $(0, 0)$, as shown in Figure 17. The existing sieve may be simply altered to incorporate the new face. First we will add the new faces,

```
addCone([(0,8), (0,1), (0,9)], (0, 11))
addCone([(0,9), (0,2), (0,10)], (0, 12))
addCone([(0,10), (0,3), (0,8)], (0, 13))
```

then the new edges,

```
addCone([(0,5), (0,7)], (0, 8))
addCone([(0,6), (0,7)], (0, 9))
addCone([(0,4), (0,7)], (0, 10))
```

noticing that the vertex will be added automatically. The old face may be eliminated by using `removePoint((0,0))`, or it can be retained to preserve the refinement history. For instance, we could add it explicitly to the topology sieve.

```
addSupport((0,0), [(0,11), (0,12), (0,13)])
```

However, this would probably be better preserved in another "refinement" sieve or a stack so that all faces continue to possess height zero.

Another common operation is the edge flip, often used in Delaunay triangulation algorithms. Here the edge dividing two triangles is removed, and a new edge is inserted joining the opposite vertices, as shown in Figure 18. Two operations on the sieve are necessary to perform the flip. First, the cone of the flipped edge becomes the two opposite vertices of the quadrangle, in this case (0,7) and (0,10). Next, the triangle cones must swap a pair of opposite edges, in this case we choose

(0,3) and (0,6). If we had chosen the other pair,(0,2) and (0,5), it would merely have placed (0,0) on the bottom. This latter change is more easily carried out by adding and deleting the individual arrows rather than by setting whole cones.

7.2 Constructing the Dual Graph

An example illustrating the power and simplicity of the Sieve construct is the formation of the essential part of a dual mesh, or element connectivity graph, in parallel. This is typically required to compute a mesh partition using graph partitioning algorithms such as those implemented in ParMetis [Karypis et al. 2005]. Consider again the doublet mesh divided between two processes, shown in Figure 2. As evident from the serial sieve on the left of Figure 2, the two simplices (0,10) and (0,11) are adjacent as they share the edge $(0, 7)$. In the partitioned sieve on the right, however, this determination cannot be made on either process separately.

To make the adjacency determination in parallel, we use the support completion operation. We only use the part of the parallel doublet mesh sieve from Figure 2 (right) representing edges and triangles, as it is sufficient for triangle adjacency determination. Clearly the local support for $(0, 3)$ contains a single element. However, we can use `supportCompletion` and `add` to construct the completed sieve on each process, shown at the top of Figure 19; the completion portion is dashed. Both processes now contain the two triangles (0,6) and (1,5) covered by a single edge $(0, 5)$. The edge separates adjacent triangles, and this fact is recorded in the triangle adjacency graph, illustrated at the bottom of Figure 19. This process can be carried out for any mesh, producing the triangle connectivity graph. Other types of connectivity data structures can be computed in an exactly analogous manner. In particular, using completion methods we can compute the dual mesh with an edge between the two members of any face with support of size two[8]. Figure 20 contains Python code to construct the dual mesh. Notice that no part of this code refers to the dimension of the mesh or any element, nor to the shape or connectivity of any element, and thus will work for any general mesh.

7.3 Partitioning in Parallel

Since partitions are merely collection of mesh elements, we may view a partition as covered by the elements it contains, and thus enlarge our sieve to include the partitions themselves. The cone of each partition point is now the set of elements in that partition. Using this construction, we may redistribute the mesh using only the cone completion operation.

First we consider the case of distributing a serial mesh that has been partitioned. The initial process will contain the full mesh sieve and an assignment of elements to processes in a partition sieve. The other processes will add the partition node corresponding to that process, for example, the prefix given by the size of the communicator and index by the process rank as in the top diagram of Figure 21. In the case of the doublet mesh used in the figure, it is sufficient to partition the triangles – elements of highest dimension – into disjoint sets corresponding to processes. Then all of the covering elements of each triangle are added to the corresponding set, possibly duplicating lower order elements on different processes.

[8]Recall that in a mesh sieve each cell of codimension 1 can be shared by at most two other cells.

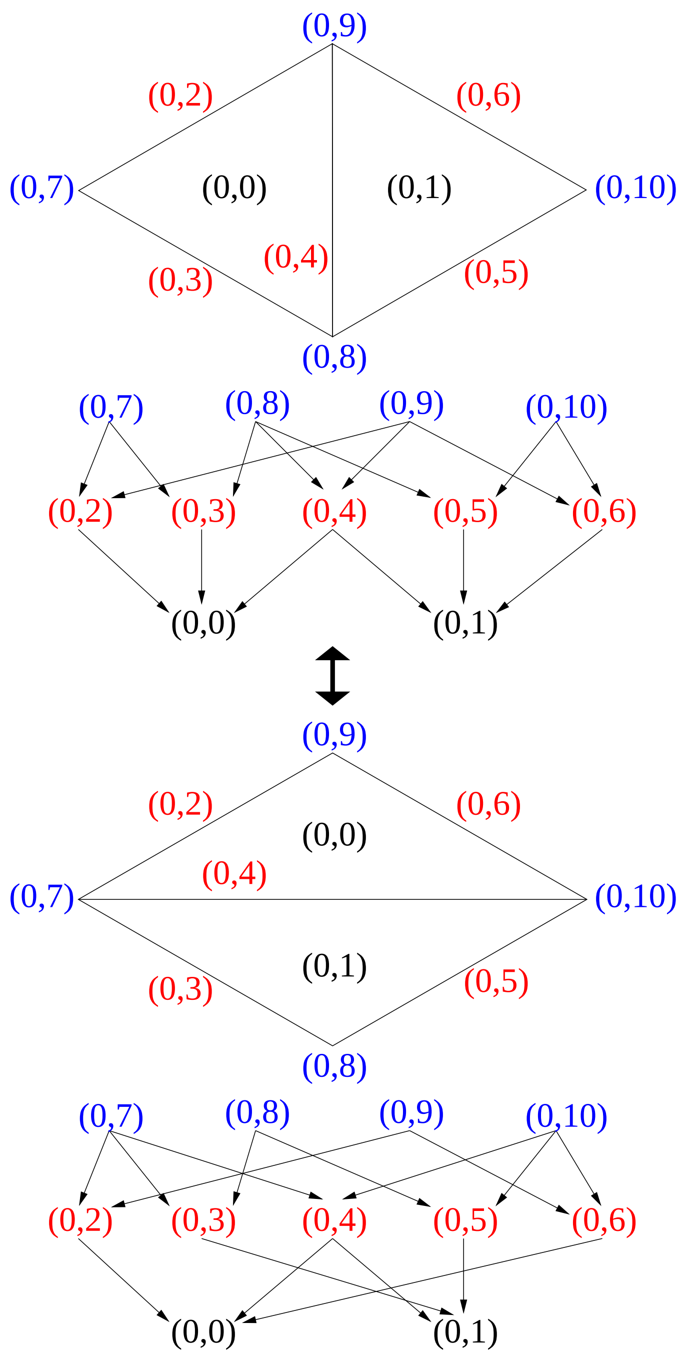

Fig. 18. The edge (0,4) is flipped.

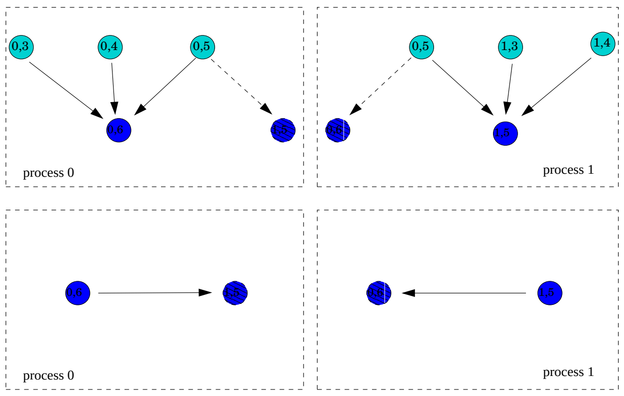

Fig. 19. One stage in the construction of a mesh connectivity graph from a (partial) distributed mesh sieve. Without completion adjacency of nodes (0,6) and (1,5) cannot be derived.

```
dualTopology = ALE.Sieve.Sieve()
dualTopology.setComm(comm)
completion, footprint = topology.supportCompletion(footprintTypeSupport)
for face in topology.heightStratum(1):
  support = topology.support(face)
  if len(support) == 2:
    dualTopology.addCone(support, face)
  elif len(support) == 1 and completion.capContains(face):
    dualTopology.addCone((support[0], completion.support(face)[0]), face)
```

Fig. 20. Python code to create the dual mesh

Such a partitioning strategy is sufficient to retain all information about a serial sieve after its distribution.

After the initial setup, cone completion will transfer the entire cone of each partition element to the remaining processes, which consists of all elements in that partition, and that completion is merged into the sieve (see middle diagram in Figure 21); on all processes other than root, the cap of the completed sieve is a duplicate of the base of the topology sieve. A second completion will transfer all the covering relations to the other processes. Finally, the local sieve on the root process is pruned of all elements whose support does not contain its partition element (see the bottom diagram in Figure 21). In the case of fully parallel rebalancing, each process creates partition nodes and cones exactly as the first process in the serial case, and then the partitioning proceeds exactly as in the serial case. Notice that the code in Figure 22 is again independent of dimension and element connectivity.

Upon cone completion, the full topology will be available on all processes; however, the element ownership will still appear as it was prior to the partition. We can renumber the elements to reflect the partition by completing the sieve once more, thereby putting all shared elements into the completion. Then a simple tie-breaking rule can be used to divide up shared elements. This method will be used to create variable numberings in Section 7.5; however, element identity is immaterial for this

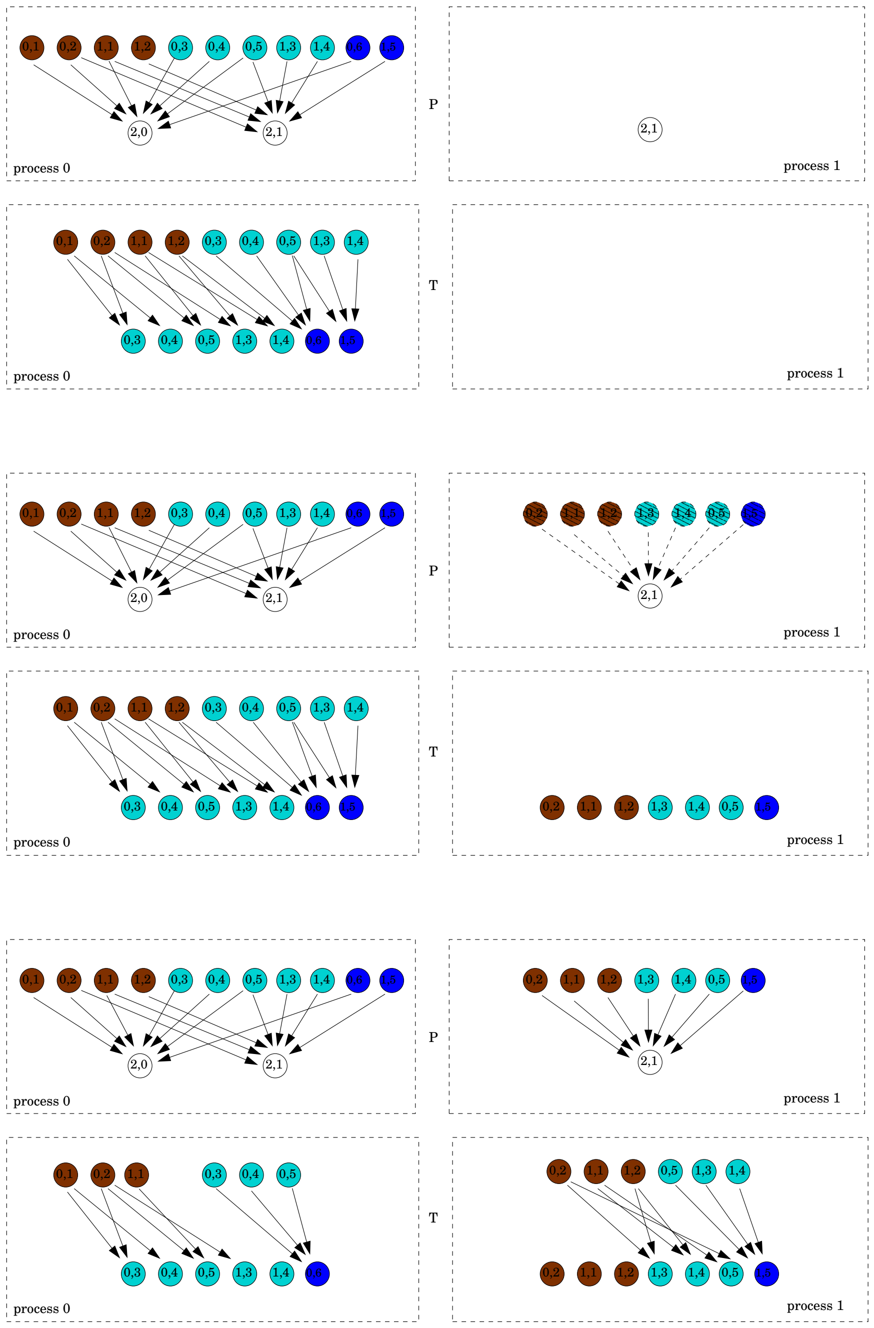

Fig. 21. Three states of sieves involved in doublet mesh partitioning. Initial state of the partitioning (P) and the topology (T) sieves (top); after completion of P (middle); after completion of T (bottom).

```
completion, footprint = topology.coneCompletion(footprintTypeCone)
topology.add(completion)
for point in topology.cone((-1, rank)):
  topology.addBasePoint(point)
completion, footprint = topology.coneCompletion(footprintTypeCone)
topology.add(completion)
topology.restrictBase(topology.cone((-1, rank)))
```

Fig. 22. Python code to partition a mesh

so we will not bother to carry it out.

7.4 Meshing in Parallel

The strategy for meshing in parallel is now quite similar to that for partitioning. We use a serial mesh generator to generate a mesh on each individual process, using an initial boundary communicated to all processes. These meshes may disagree on the boundary; however, we will construct a cover of the original edges by any edges produced by splitting. Then, using cone completion, we can construct a complete covering of the boundary. We must then resolve incommensurate coverings of each boundary edge either by merging two close points or by introducing another edge between the two. Any newly introduced boundary edges are incorporated into the existing process mesh. As a final step, we may run a parallel mesh improvement algorithm, without topology change, to smooth out any distortions caused by merging [Munson 2004]. This procedure will result in a mesh that may be artificially refined along process boundaries, but this effect becomes negligible as the mesh grows, and we would argue that it is far outweighed by the simplicity of the algorithm itself.

We also note that a serial mesh generator can be used to construct periodic meshes. Just as in the parallel mesh example, we must merge a boundary, in this case the identified periodic boundary. This merge is accomplished in exactly the same manner, resulting in a fully periodic mesh.

7.5 Constructing a Finite Element

We have remarked above that the restriction and prolongation operations are exactly the mechanism responsible for finite element assembly. Here we demonstrate this in detail for a higher-order Lagrange element on multiple fields. The restriction of a continuum field to an element $\mathcal{T}$ is what we approximate using our finite element space P. For our Lagrange element, we use a *nodal* basis, meaning that the basis $\{\phi_i\}$ for P and the basis of functionals $\{L_j\}$ for its dual P' satisfy

$$L_j(\phi_i) = \delta_{ij}.$$

Thus, the restriction process can be thought of as a selection of the coefficients for basis functions in P that are nonvanishing on $\mathcal{T}$, and each coefficient can be identified with the particular L_j which does not annihilate that basis function. Furthermore, each L_j is identified with a certain topological piece of the element. In the case of point evaluations, this is merely the piece on which the evaluation point occurs.

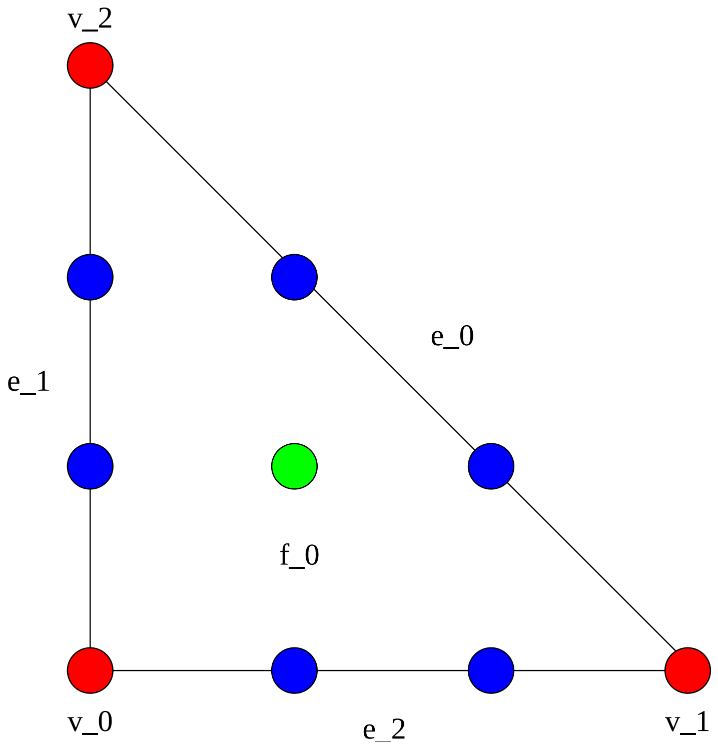

Fig. 23. Third-order Lagrange element

Consider the third-order Lagrange element on a triangle. In Figure 23, the circles represent the point evaluation functionals that form a basis or P'. If we, for instance, restrict to e_2, then we will require the four coefficients associated with (v_0, v_1, e_2). Restricting to $\mathcal{T}$, we need the coefficients from all the labeled topological elements, which we can recognize from above as the cone of f_0.

In traditional finite element code, one represents the discretized field by an array of basis function coefficients. This approach forces the user to construct a numbering on the basis functions and then select the correct indices for each element. Thus, the code is more complicated, and its relation to the mathematics is obfuscated. In our paradigm, we replace the integers in indexing with the topological elements themselves, and selection occurs through the cone construction. In addition, when we consider the problem of representing multiple discrete fields over the mesh, we realize the full power of the Stack as lattice operations are incorporated. We can represent degrees of freedom as points in a discrete sieve (i.e., a sieve without arrows). Variables may be associated with topological elements via vertical arrows with the topology as the base. However, we would like a mechanism to segregate variables in each field. Each field is represented by a point in an auxiliary sieve, which is then used as the base of a stack whose vertical arrows connect to the degree of freedom sieve. The cone over each field point is the set of variables in the field,

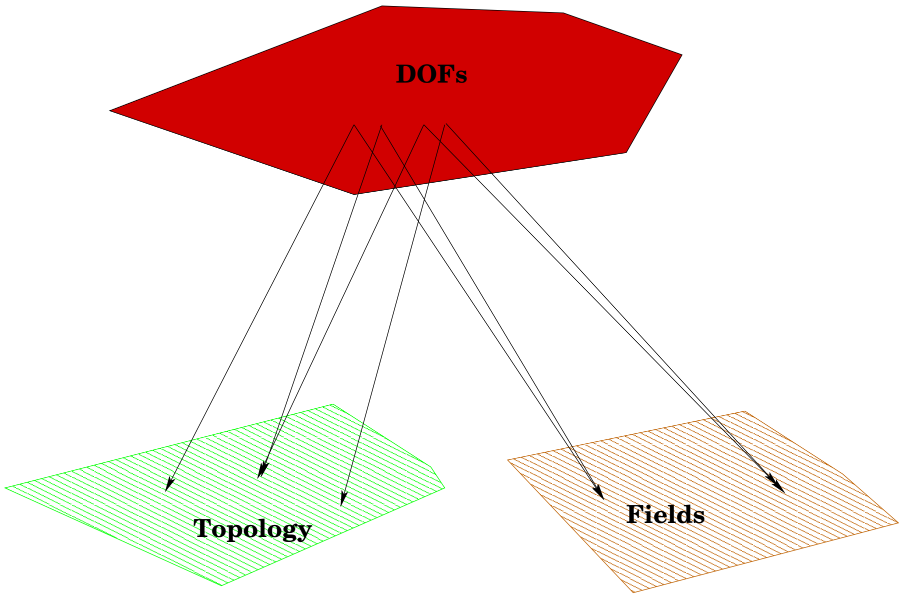

Fig. 24. Illustration of degrees of freedom for multiple fields

```
elements = [FIAT.Lagrange.Lagrange(FIAT.shapes.TRIANGLE, 2),
            FIAT.Lagrange.Lagrange(FIAT.shapes.TRIANGLE, 3)]
ranks = [1, 0]
dim = mesh.getDimension()
dof = ALE.Sieve.Sieve()
dof.setComm(comm)
numbering = ALE.Stack.Stack()
numbering.setComm(comm)
numbering.setTop(dof)
numbering.setBottom(topology)
```

Fig. 25. Initial specifications for the finite elements

as depicted in Figure 24, and the degrees of freedom from a given field over a given element are simply the meet of the cones in the corresponding base points. This approach also allows us to order the degrees of freedom in a manner suitable for a given application, or even reorder them for different stages of the computation, without affecting the retrieval code.

As a concrete example, we create a numbering for two fields over the doublet mesh. The first is a P_2 vector field and the second a P_3 scalar field. Figure 25 shows these initial choices and the creation of the numbering stack. In Figure 27 we present the numbering algorithm itself.

We loop over all elements in the topology, and if the element is present on more than one process, only the lowest-rank process will create degrees of freedom over it. We then loop over each field and decide how many degrees of freedom the

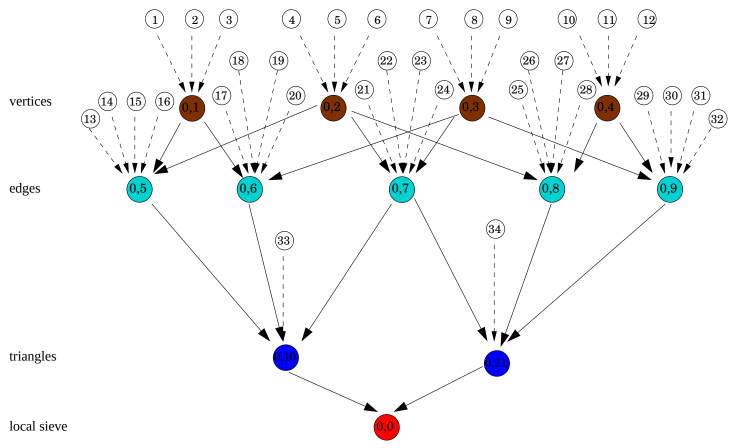

Fig. 26. DOF Stack for the trial discretization over a doublet mesh

```
completion, footprint = topology.supportCompletion(footprintTypeSupport)
index = 0
for p in topology.space():
  if completion.capContains(p):
    neighbors = footprint.support([p]+list(completion.support(p)))
    if [0 for processTie in neighbors if processTie[1] < rank]:
      continue
  indices = []
  for field in range(len(elements)):
    scalarDof = len(elements[field].dual_basis().getNodeIDs(topology.depth(p))[0])
    entityDof = scalarDof*max(1, dim*ranks[field])
    if entityDof:
      var = [(-(rank+1), index+i) for i in range(entityDof)]
      indices.extend(var)
      index += entityDof
      dof.addCone(var, (0, field))
  numbering.addCone(indices, p)
completion, footprint = numbering.coneCompletion(footprintTypeCone)
```

Fig. 27. Python code to create a variable ordering

field possesses on that element. FIAT can provide the size of the dual space on an element of that dimension, and we equate dimension with depth in the sieve. Finally, we assume that tensor fields have dimension equal to the dimension of the mesh. We add these new degrees of freedom to the cone of the current field and then add the collective degrees of freedom for the current element to its vertical cone in the numbering stack. Lastly, the stack is completed over vertical arrows so that ghost degrees of freedom are available. The full stack is shown in Figure 26.

If we let f and K be the local element vector and matrix, F a global discrete field, and J its Jacobian, the code in Figure 28 will assemble the function and its Jacobian.

Using both sieves and stacks, we have been able to provide routines that partition

```
elements = mesh.topology.space()
elemU = u.restrict(elements)
# Loop over highest dimensional elements
for element in mesh.topology.heightStratum(0):
  # We want values over the element and all its coverings
  chain = mesh.closure(element)
  # Retrieve the field coefficients for this element
  coeffs = elemU.getValues([element])
  # Calculate the stiffness matrix and load vector
  K, f = self.integrate(coeffs, self.jacobian(element, mesh, space))
  # Place results in global storage
  elemF.setValues([chain], f)
  elemJ.setValues([[chain], [chain]], K)
F = elemF.assemble([])
J = elemJ.assemble([])
```

Fig. 28. Python code to assemble the linear system. The sifting policy identifies elements with their interiors and admits closure chains as input denoting the corresponding geometric closure.

a mesh in parallel, calculate the finite element variable ordering, integrate the weak form, and assemble the operator. These routines are independent of the mesh dimension, global topology, element shapes, and finite element.

8. CONCLUSIONS

A key conclusion of this effort is that better mathematical abstractions in software bring concrete benefits. In current FEM simulation packages, reusability rarely goes beyond linear algebra level. The reason is a lack of effective mathematical abstractions for hierarchically structured data and operations that adequately reflect the modeled problem. Components cannot be shared when operations are inextricably linked in the implementation. Furthermore, the complexity of the existing hierarchical solvers written without the benefit of these abstractions increases very quickly, creating inpenetrable and unmaintainable code. This low-level approach to implementation has also greatly hindered generalization of these algorithms, for instance to regions with nontrivial global topology.

However, many of these difficulties can be rectified by using the Sieve construct and sieved array structures. Discarding the explicit dimensionality and shape information in the algorithms not only reduces the complexity but also results in much greater generality. All operations are expressed in terms of a single covering relation, which captures the ubiquitous notion of a decomposition of objects into more elementary parts. With only a single routine with parallel communication, optimization and portability also become much easier. Furthermore, since this approach assumes much less about the structure of the problem, sieves can be more easily incorporated into existing PDE frameworks, and perhaps frameworks for other problems as well. Moreover, the generality of the interface enhances the capabilities of existing PDE solvers. Sieves can seamlessly handle hybrid meshes, complicated global topologies as in micromagnetics, and intricate structures embedded in the mesh such as fault systems in seismic modeling.

9. GLOSSARY

chain: A set of sieve points, often identified with a set of topological mesh elements.

cochain: A set of points in a dual sieve, often identified with a set of functions over toplogical mesh elements.

cocycle: A cochain satisfying consistency conditions on the meet of the elements, often identified with the agreement along element intersections of functions defined on the elements.

covering: A relation between two sieve points, expressed by a sieve arrow. The notion can be extended to a relation between two chains.

complete covering: A chain covering which includes every member of the cone of each member of the base chain.

localSieve: An artificial sieve point added to cover all the leaves on the process.

sieve: A directed acyclic graph which expresses a covering relation between points, meant to encode an *étale* topology.

stack: A sieve which has another sieve for both its cap and base. Sieve operations operate only over arrows between these sieves, not inside them.

sieved array: Storage structured according to the covering relations of a sieve.

totalSieve: An artificial sieve point added to cover all the leaves in the sieve.

ACKNOWLEDGMENTS

We wish to thank Barry Smith for many helpful discussions and suggestions for improving the paper; Robert Kirby for careful readings of the drafts; and Gary Miller for useful comments.

REFERENCES

Aleksandrov, P. 1957. *Combinatorial Topology.* Graylock Press, Rochester, N. Y.

Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. 2005. PETSc Web page. `http://www.mcs.anl.gov/petsc`.

Barr, M. and Wells, C. 1985. *Toposes, Triples and Theories.* Springer, Berlin.

Brisson, E. 1989. Representing geometric structures in d dimensions: Topology and order. In *SCG '89: Proceedings of the Fifth Annual Symposium on Computational Geometry.* ACM Press, New York, NY, USA, 218–227.

Brown, D., Glimm, J., and Freitag, L. 2005. Terascale Simulation Tools and Technology (TSTT) Center. `http://www.tstt-scidac.org`.

Ciarlet, P. 1978. *The Finite Element Method for Elliptic Problems.* North-Holland.

Dupont, T., Hoffman, J., Jansson, J., Johnson, C., Kirby, R. C., Knepley, M., Larson, M., Logg, A., and Scott, R. 2005. FENICS Web Page. `http://www.fenics.org`.

F. Chung, R. Andersen, L. L. 2004. Drawing power law graphs. In *12th International Symposium on Graph Drawing.*

Fitch et. al., B. G. 2005. Blue matter: Strong scaling of molecular dynamics on BlueGene/L. Research Report W0508-035, IBM Research. August.

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

Kirby, R. C. Optimizing FIAT with Level 3 BLAS. *ACM Transactions on Mathematical Software.*

KIRBY, R. C. 2004. Algorithm 839: FIAT: A new paradigm for computing finite element basis function. *ACM Transactions on Mathematical Software 30,* 4 (Dec.), 502–516.

KLAWONN, A., RHEINBACH, O., AND WIDLUND, O. B. 2005. Some computational results for dual-primal feti methods for three dimensional elliptic problems. In *Domain Decomposition Methods in Science and Engineering*, R. K. et. al., Ed. Lect. Notes Comput. Sci. Eng., vol. 40. Springer, Berlin, Germany, 361–368.

KNEPLEY, M. AND KARPEEV, D. A categorial approach to computational meshes. in preparation.

KNEPLEY ET AL., M. 2005. Argonne SIDL Environment Web page. `http://www.mcs.anl.gov/ase`.

MPI FORUM. 2005. MPI forum home page. http://www.mpi.org.

MUNSON, T. 2004. Mesh shape-quality optimization using the inverse mean-ratio metric. Preprint ANL/MCS-P1136-0304, Argonne National Laboratory. April.

SHEWCHUK, J. 2005. Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. `http://www-2.cs.cmu.edu/~quake/triangle.html`.

SI, H. 2005. TetGen: A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator. `http://tetgen.berlios.de`.