A NEW SOFTWARE FRAMEWORK FOR UNSTRUCTURED MESH REPRESENTATION AND MANIPULATION

By

Craig Robert Tanis

Approved:

William K. Anderson
Professor of Computational Engineering
(Chair)

Steve Karman
Professor of Computational Engineering
(Committee Member)

Sagar Kapadia
Assistant Research Professor
of Computational Engineering
(Committee Member)

John Matthews
Associate Professor of Mathematics
(Committee Member)

A NEW SOFTWARE FRAMEWORK FOR UNSTRUCTURED MESH REPRESENTATION AND MANIPULATION

By

Craig Robert Tanis

A Dissertation Submitted to the Faculty of the University
of Tennessee at Chattanooga in Partial Fulfillment
of the Requirements of the Degree of
Doctor of Philosophy in Computational Engineering

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

December 2013

ABSTRACT

This research presents a unique new software framework for representing and manipulating unstructured meshes in parallel, for use in modern scientific simulation codes. Due to the central nature of the unstructured mesh, this framework provides a variety of functionality, desirable throughout the lifecycle of an application, such as IO, parallel partitioning, phantom node data updates, adaptive refinement, derefinement and load balancing.

What makes the framework unique is a focus on generality: like a database, the user provides a programmatic schema defining the structure of the mesh, including topological descriptions of the valid mesh entities. The system extracts adjacency information from this input and allows the use of high-level queries for manipulating and processing the mesh. Advanced C++ techniques allow for a combination of high extensibility and highly optimizable code.

New applications can be built quickly, by taking advantage of the framework's capabilities. Existing codes can incorporate the framework with minimal modification, due to the use of data proxies that mediate between the framework's internal data structures and existing user data.

The design and implementation of this framework are discussed, and several representative applications are presented. Scalability results and analysis are included.

DEDICATION

To my beautiful wife: you are my heartbeat and my breath. You are the kick in my butt.

To my beautiful children: you are the goofiest and most brilliant kids on the planet.

We did it!

ACKNOWLEDGMENTS

I gratefully acknowledge the indispensable support and direction provided by my advisor, Dr. Kyle Anderson.

I also thank Dr. Sagar Kapadia for providing and patiently supporting the 3D flow solver used in this work.

To my entire committee: thank you for your contributions to this paper, and to the project as a whole.

To my colleagues and friends throughout the SimCenter, the Computer Science Department, and UTC at large: thank you for the encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF LISTINGS

CHAPTER 1
INTRODUCTION

Computational simulations based on the numeric (approximate) solution of partial differential equations require a discretization of the solution domain over which to calculate values. *Unstructured meshes* are a historically popular choice for this discretization, as these meshes can easily be generated for complex geometries; and can, furthermore, be locally modified (or *adapted*) as a simulation progresses and higher mesh resolution is deemed necessary in specific regions of the domain [1].

This work explores a new general data structure and code library, *Splatter*, for representing unstructured meshes in MPI-parallel simulation codes. It is designed to efficiently represent arbitrary mesh entities, with associated data, and it supports mesh partitioning, adaptation (refinement and coarsening), and load-balancing.

A unique feature of *Splatter* is its highly flexible query syntax. The user may embed queries (such as for entity adjacency searches or topological feature extraction) directly in C++ code. Using standard C++ *metaprogramming* techniques, the compiler turns *Splatter* queries into optimized C++ code. This feature sets *Splatter* apart as an ideal candidate for experimental applications.

In Chapter 2, the need for computational meshes is discussed, as well as the value of general mesh-management frameworks. We discuss several modern mesh-management packages, and motivate the development of *Splatter* in the context of modern scientific software engineering.

In Chapter 3, the design of *Splatter* is discussed, with a focus on the design concepts that make the framework unique. Specifically, this includes a discussion on mesh entity adjacency and the definition of mesh entity types, the use of data proxies in a distributed memory application, and an overview of the framework's embedded query syntax.

In Chapter 4, the C++ implementation of key framework components is covered, with a focus on C++ techniques for optimized performance. An analysis of key algorithms is also presented.

In Chapter 5, several applications of the *Splatter* framework are demonstrated, that take advantage of its unique architecture. Specifically, parallel partitioning, dynamic adaptive refinement and de-refinement and load-balancing are discussed. Additionally, the process of modifying an existing 3D computational flow simulation to use *Splatter* in presented. With minor modifications, an existing code can take advantage of advanced global mesh manipulation operations.

Finally, in Chapter 6, the results of this research are summarized, concerning the value of *Splatter* for parallel mesh management in scientific simulations. In closing, some plans for future work are suggested, including modifications to the *Splatter* framework for use in application codes consisting of multiple programming languages, and potential improvements to the query functionality.

CHAPTER 2
BACKGROUND

2.1 Unstructured Meshes

Computational simulations often involve the repeated solution of partial differential equations over some analysis domain. These computed solutions are inherently approximations, based on a *discretization* of both the underlying equations and the analysis domain.

The domain discretization provides specific points in space (and potentially time) at which solutions are determined. The point is to replace the continuous, infinitely resolvable physical domain with a finite number of calculable points.

Consider the simulation of air flow around an airfoil. This is a classic scenario, in which the scientist is interested in determining the density, pressure and velocity of air in the surrounding region, from which additional values such as lift may be obtained. Figure 2.1 illustrates an *unstructured* discretization suitable for such a simulation, in two-dimensions.

The simulation may proceed by approximating a solution of the *Navier-Stokes* equations at the vertices of this discretization, using information about the adjacent triangles to determine the volume that each vertex is representing.[1]

The domain in this example has been completely subdivided into non-overlapping triangles, preserving the volume of the original geometry as well as possible, within the bounds of numeric error. Similarly, the surface of the airfoil has been discretized into straight-line edges that match up to the triangles along the boundary. This volume-conserving collection of spatial subdivisions, and corresponding boundary edges is called a *computational*

[1]Different mathematical interpretations of this spatial discretization may be used, as long as they are compatible with the chosen discretization of the equations being solved.

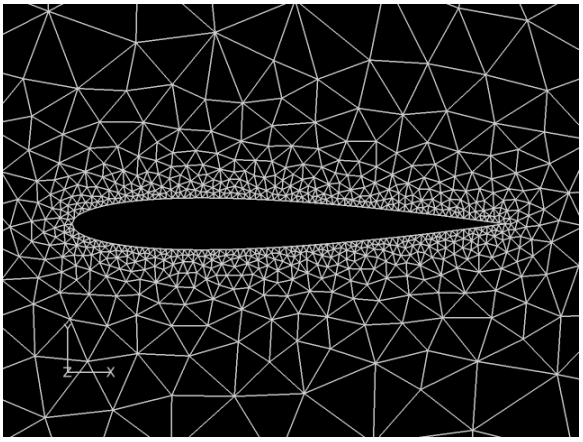

Figure 2.1 Unstructured spatial discretization for a 2D fluid flow simulation.

mesh. It consists of a collection of *mesh entities* (in this case, the triangles and boundary edges).

This mesh is considered *unstructured* because there is no clear ordering of the mesh entities — that is, the connectivity between mesh entities is irregular [2]. This is in contrast to a *structured* mesh in which there is an implicit ordering of the mesh entities, due to an ordered set of refinements along each principal axis (a grid).

Conceptually, an unstructured mesh could contain any polytopal [2] entity types. The example in Figure 2.1 is, formally, a *simplicial complex* covering the domain [3], because it consists entirely of simplices (triangles, tetrahedra). A mesh need not be simplicial, however, and can easily involve more complex volume and surface elements, resulting in a general *polytopal complex* [4][3].

The choice of entity types to use in a mesh is mostly governed by implementation details of the application. Many solvers are developed to only work with simplices, but certain applications benefit from mixed types, such as using packed hexahedral entities along internal boundaries to resolve viscous effects in a flow simulation [5]. Pyramids and prisms can be used as transitional entities between the quadrilateral and triangular faces.

A mesh involves more than just geometric data. From a mathematical standpoint, the partial differential equations (*PDE*'s) solved over a mesh require boundary conditions.

[2] n-dimensional polygon with flat sides

Different boundaries may impose different boundary conditions, and so a mechanism must be in place to associate mesh entities along boundaries with the appropriate conditions.

From the perspective of mesh management, the simplest implementation of multiple boundary conditions requires a simple integer flag variable associated with all boundary entities indicating the equation type to use for those entities. More complex boundary conditions may require the calculation of quasi-solution variables to store transient data along the boundary.

This concept is extended to volume entities as well, where different regions in the domain have different solution properties. For example, in [6] — a solid-oxide fuel-cell simulation — volume conditions distinguish between free flow regions and those in the electrode, which correspond to materials with a different porosity, motivating different flow equations for those particular regions. This particular simulation also identifies boundaries along which chemical reactions occur, which require the storage and processing of additional values.

As a rule, larger meshes yield higher quality application results. These larger meshes — larger, meaning more vertices and more entities of all relevant dimensions — demand increasingly more computational resources. Modern mesh-based applications can involve the use of many tens of thousands of individual processors, collectively using their memory to store mesh data and share in the computational load.

In this work, we develop a new software framework, *Splatter*, useful for the representation and manipulation of general unstructured meshes in parallel. The goal is to encourage application scientists to focus on the mathematics and physics of the problem they are solving, and use this framework to handle the mesh, and all of its related complexities.

2.2 Mesh Frameworks

The concept of a mesh management framework is not a novel one. After all, parallel programming is "hard" [7] and application scientists of the world would greatly benefit from the existence of a reusable system for dealing with parallel unstructured meshes.

Other frameworks exists, and have inspired and directed the development of *Splatter* in key ways.

FMDB (Flexible Mesh DataBase) is a mesh management framework developed at Rensselaer Polytechnic Institute (RPI). Originally produced as part of Dr. Seegyoung Seol's dissertation work [8], FMDB is based on topological concepts originally documented by Beall and Shephard (Dr. Seol's PhD. advisor) in their seminal 1997 work [9]. FMDB underlies much of RPI's considerable computational research output, including research on mesh adaptation [10, 11], entity reordering [12] and component-based high-performance computing [13, 14].

FMDB enables adjacency traversal and mesh manipulation in parallel, but the interface is a highly non-intuitive C API [15]. It has been criticized as using excessive memory [16]. Despite these subjective shortcomings, FMDB encompasses many key ideas, and is under active development.

Additionally, FMDB provides an ITAPS (Interoperable Technologies for Advanced Petascale Simulations) interface [17], which is a movement to formalize the interaction between reusable scientific software components. This is a subject of future work for *Splatter*.

MOAB (A Mesh-Oriented datABase) [18] is an open-source mesh-management framework from Argonne National Laboratory. It uses a reduced mesh representation to store entity adjacency. In addition to representing unstructured grids, it also supports structured grids. It has limited support for mesh modification [2].

The fact that MOAB does not support mesh modification well means that it could not be considered a substitute for other frameworks listed, but its use of a reduced mesh representation and corresponding search structures are intriguing ideas.

MSTK [19] is a mesh library from Los Alamos National Laboratory that is configurable to use either full or reduced mesh representation for storing mesh entities, and it supports general polygons and polyhedra. It does not, however, support mesh modification in parallel, so the user is limited by serial processing capabilities.

Notable mesh-management components exist in a number of full featured numerical solution frameworks. The Trilinos project from Sandia National Laboratory includes a mesh framework called STK (Sierra Toolkit Mesh) [20]. It supports general adjacency retrieval, as well as mesh refinement and load-balancing. Documentation is minimal, however [2], and it is clearly intended for use with other Trilinos packages. Similarly, libMesh [21] is a numerical framework originally developed at the University of Texas, that has a separate mesh-management subpackage that supports parallel mesh modification and load-balancing. It is, again, intended to be used along with the rest of the libMesh framework and may not be an appropriate choice for general codes.

This sampling of existing mesh frameworks includes some inspiring, powerful systems, but they require a significant commitment from their users. Integration of existing applications with these frameworks is difficult, due in no small part to their cryptic APIs and very specific usage patterns.

The goal behind *Splatter* is to provide comparable performance to these larger tools, in a manner that is owned by the user. It's a framework that's not a framework — it's an unobtrusive core kernel of functionality required for processing distributed unstructured data, along with a variety of options for configuration and integration, and – perhaps most interestingly – an expressive embedded programming language for abstract mesh manipulation.

Over these chapters, it will become clear that the framework is extensible, it is unique, and it works.

CHAPTER 3
DESIGN CONCEPTS

3.1 Overview

Splatter is a software framework primarily intended for the representation and manipulation of arbitrary unstructured meshes in parallel scientific simulation codes. Since the mesh is the centerpiece of many (typical) simulation code operations, the framework serves a variety of functions for use throughout the process.

Splatter's primary responsibility is simply to store the mesh and any associated data in a way that allows users to iterate over the mesh elements and perform arbitrary calculations. In parallel, *Splatter* imposes a fairly standard mesh partitioning scheme, and handles all inter-process mesh data synchronization. In more advanced applications, *Splatter* allows dynamic topological changes to occur, such as adaptive refinement (and de-refinement), and is capable of dynamically redistributing the mesh among processors in order to better balance the load, in response to these topological changes.

We will see in Chapter 5 the process by which *Splatter* functionality can be introduced to existing applications with minimal modification. This simplicity is due, in part, to the framework's unique data model based on *proxies*. Mesh manipulation operations are written in terms of these proxies which are responsible for modifying associated user data structures.

Splatter is designed to be extremely general. While most unstructured meshes are based on predictable geometric entities (tetrahedra, triangles, edges, etc.), it is conceivable that a user may want to represent *different* entities – either unconventional geometric entities, alternate representations of conventional entities or something more abstract and unexpected. To handle this, *Splatter* views the mesh like a database. The user provides, essentially, a schema describing the format of allowed mesh entities, and writes *queries* that define mesh operations.

This design provides powerful capabilities that enable interesting applications. Section 5.3 describes an advanced use of this query functionality in the management of parallel bookkeeping information, supporting adaptive refinement and de-refinement.

3.2 General Mesh Representation

An unstructured mesh is, generally, a polytopal complex — an arrangement of non-overlapping, *face-sharing* polytopes (polyhedra in 3D, polygons in 2D) — resulting from some *mesh generation* process. A mesh generation process involves the spatial decomposition of an analysis domain, and the resulting complex is a volume-conserving discretization of the analysis domain's geometry. For numeric purposes, the resulting polytopes are usually convex; for simplicity's sake, the resulting polytopes are often simplices (tetrahedra, triangles). Figure 3.1 illustrates a 2D *polygonal* complex consisting of quadrilaterals, triangles, edges, and vertices.

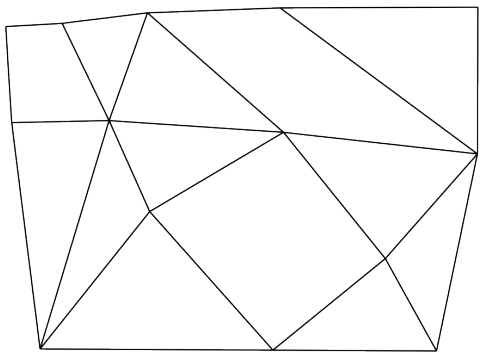

Figure 3.1 A polytopal complex in 2D

Splatter can represent any polytopal complex, given the following constraints:

1. Each vertex in the polytope must be assigned a unique integer id.
2. Each polytope in the complex must be identified by a TYPE tag and an ordered tuple of its vertex ids.

The TYPE tag corresponds, typically, to the geometric classification of the polytope, which, in turn, provides a geometric interpretation of the node tuples.

Figure 3.2 illustrates how a 4-node tuple can be interpreted as a quadrilateral or a tetrahedron. The associated TYPE tag specifies which interpretation is valid. Note: this interpretation is strictly concerning the topological relationship between these nodes, and does not involve any spatial coordinates associated with the nodes.

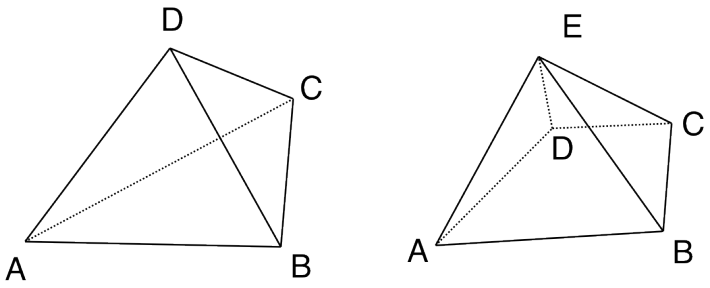

Figure 3.2 4 nodes could be a quadrilateral or tetrahedron

Modern mesh generation software outputs unstructured meshes in a format compatible with these requirements [22]. In fact, The CFD General Notation System (CGNS) [23] – *an AIAA Recommended Practice* – aims to standardize the representation and distribution of unstructured meshes, and does so in a manner totally compatible with the above constraints.

Typically, a mesh generation process outputs the unstructured mesh as a collection of homogeneous lists of polytopes as tuples (each list consisting of a single TYPE – a list of triangles, a list of tetrahedron, a list of prisms, etc.). These polytopes are often associated with domain-specific data, such as boundary or region condition flags. Vertices are numbered consecutively and associated with their spatial coordinates.

As a matter of nomenclature, we refer to these vertices as *nodes*, and all polytopes are referred to as *mesh entities*. The geometric classification (TYPE tag) associated with mesh entities are *mesh entity types*.

3.2.1 Mesh Entity Types

The mesh entity type imposes a geometric interpretation on a tuple of nodes. This type is consulted whenever a geometric operation must be performed, such as calculating entity volume or determining neighboring entities (adjacency search).

The type defines a tuple size, as well as a *winding* of the nodes. Given the tuple and tag, the original polytope can be constructed.

Consider a pyramid. The standard *Splatter* entity type representing a pyramid (defined in Section 4.2.1.3) imposes the following geometric properties (see Figure 3.3):

1. The tuple has 5 nodes.

2. The first four nodes correspond to the square base of the pyramid, ordered such that when following the *right-hand rule* the thumb is pointing at the 5^{th} node, or peak of the pyramid.

This definition imposes a structure and order on the nodes contained in a pyramid tuple.

Furthermore, this geometric interpretation embeds adjacency information — in essence, answering the question "how can pyramids be positioned relative to other entities in a mesh?"

3.2.1.1 Adjacency

In Section 3.2, *polytopal complex* was defined as an arrangement of face-sharing polytopes. *Adjacency*, for the purposes of this work, is the relationship between two polytopes that share a face [9]. Adjacency information is highly valuable during feature extraction and mesh processing, such as is commonly performed during mesh-based applications, and is thus an important feature of a mesh-management framework.

Following the nomenclature introduced in [9], define

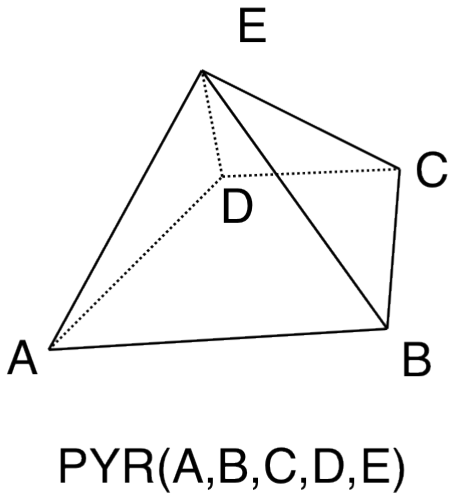

Figure 3.3 Pyramid ordering respects the right-hand rule

M	the mesh
$\{M^d\}$	an unordered list of the entities in M of dimension d
M_i^d or $\{M^d\}_i$	a specific (the i^{th}) entity in $\{M^d\}$
$\phi\{M^d\}$	the set of entities in $\{M^d\}$ contained in or adjacent to ϕ

Note: ϕ may be a single entity or group of entities. The dimension of an entity is defined by the dimension of its primary metric. Specifically dimension is 3 for entities with volume (regions), 2 for entities with area (surfaces), 1 for entities with length (edges), 0 for entities with no size (nodes or vertices).

An entity A may *contain* other entities (*subentities*) that are *downwards adjacent* to A. In that case, A is *upwards adjacent* to those subentities.

For example, $M_i^2\{M^3\}$ is the set of volume entities (M^3) upwards adjacent from the i^{th} surface entity. $M_i^3\{M_i^2\}$ is the set of surface entities contained in (downwards adjacent from) the i^{th} region entity.

Entities are *neighbors* if they are of the same dimension, and share a lower adjacency of a dimension one less. $M_i^3\{M^2\}\{M^3\}$ is the set of volume entities containing surface entities that are contained in the i^{th} region entity, in other words the neighbors of M_i^3.

In the case of a pyramid entity, the adjacent lower-dimensional entities are the constituent 2-dimensional subentities (4 triangles and 1 quadrilateral), and their 1-dimensional edges and 0 dimensional nodes.

Given a specific pyramid, Pyr, $Pyr\{M^2\}$ are the surface (2D) entities defining the polygonal faces of the pyramid — specifically 4 triangles and a quadrilateral. Region entities, other than Pyr that contain any of these would be neighbors of Pyr (Figure 3.4).

With this nomenclature, it is possible to discuss the adjacency information maintained by the *Splatter* framework, and how it may be exploited by an application.

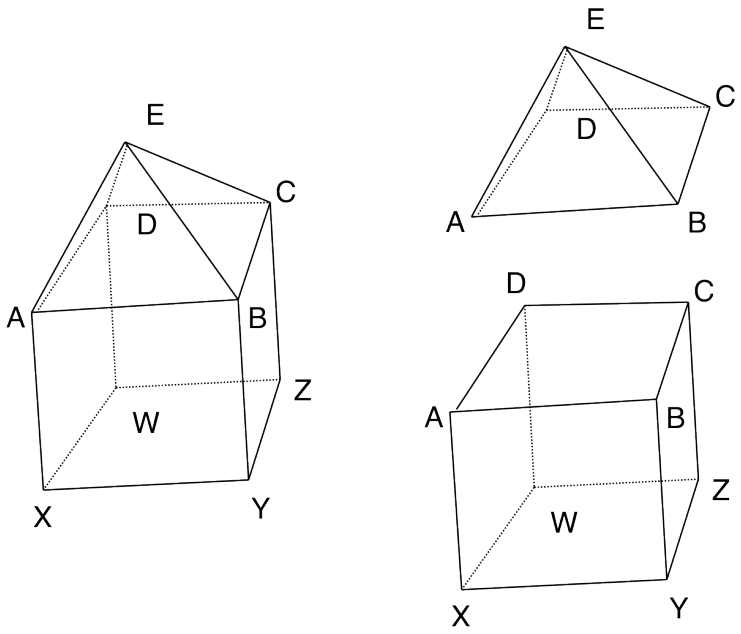

Figure 3.4 The pyramid and hexahedron are neighbors, sharing a quadrilateral

3.2.1.2 Allowed Entity Types

In *Splatter*, the user provides a programmatic definition of all valid mesh entity types (a standard set of types, containing edges, triangles, quadrilaterals, tetrahedra, pyramids, prisms, and hexahedra is part of the framework).

These mesh entity types are defined recursively, based on the adjacency concepts described above. Starting with the fundamental building block of all unstructured meshes, the 0-dimensional node, an edge can be defined as a 2-tuple of nodes with arbitrary winding. Triangles and quadrilaterals are built out of edges. Tetrahedra are built out of triangles; Pyramids are built out of triangles and quadrilaterals, etc.

In short, entities with dimension d are specified in terms of the adjacent entities of dimension $d-1$. These definitions embed enough information to support general *adjacency search* described in the next section.

This definition of allowed entity types, in conjunction with the arrangement of entity indices (see Section 3.4), effectively defines the schema for a mesh. The resulting mesh organization can be processed using user-defined queries. These concepts together yield a view of the mesh as a flexible database – one of the key concepts driving the entire *Splatter* design. Section 3.5 discusses the use of queries. Section 4.2.1.3 provides the details on entity type implementation.

3.2.1.3 Adjacency Search

To support these kinds of adjacency queries, a mesh management framework must maintain a search data structure that represents adjacency information between entities. This is a potentially complex task, especially when meshes are dynamic data structures — shifting between parallel partitions and undergoing topological transformations.

A *full* adjacency representation would store all upwards and downwards adjacencies for every entity in the mesh. If such a structure were possible, it would be very fast in that any adjacency query could be satisfied in a single access. As discussed, modern meshes are

very large (hence the need for parallel processing) and size of such a data structure would be extremely prohibitive. A *reduced* representation may use less memory at the expense of lengthier processing for certain lookups. A variety of representational options are well-presented in [9].

Splatter maintains a reduced set of adjacency relationships in memory that can be expanded to full upward and downward adjacencies in constant time — a reduced approach referred to as *circular adjacency*, also introduced in [9].

In this scheme, the only physical storage required is a map from each node to the set of containing entities with the highest dimension. (In a 3D mesh, it could be a quick lookup table mapping nodes to containing region entities).

To determine lower-dimensional entities containing a node, generate the subentities of the highest dimension using the patterns defined in the corresponding entity type. Since each subentity generation process lowers the dimension by 1, there are at most 3 such entity generations possible for a given node lookup. This is effectively constant time, on average, though some nodes may exist in a disproportionately high number of region entities. In this way, the worst case access would be $\mathcal{O}(\max(\text{entities per node}))$

More complex adjacency lookups proceed by performing set intersection operations on the relevant entities. Algorithm 3.1 presents an algorithm for determining 3D entities that contain a given 2D (surface) entity.

The *Splatter* implementation of this concept is discussed in Section 4.2.2.2. It should be noted that the framework design allows the integration of different adjacency search structures through normal C++ inheritance mechanisms. This possibility is briefly discussed in Section 4.2.2.1.

3.3 In Parallel

Modern problems of interest are posed in terms of computational meshes containing potentially many millions (or billions) of nodes and entities. As larger meshes become

```
input  : face – the 2D entity for which we search
input  : hash – the search structure mapping nodes to sets of region entities
declare: n – an integer
declare: regions – the working set of 3D region entities

look up the set of regions containing the 0^th node in face
n ← 0
regions ← hash[ face.node[n] ]

while regions ≠ ∅ and n < face.num_nodes do
    regions ← regions ∩ hash[ face.node[n]]
    n ← n + 1
end

output : regions
```

Algorithm 3.1 Determining 3D entities containing a given 2D entity

commonplace, the memory demands of the systems that process them are increased, and the algorithms that process the meshes take increasingly longer times.

To accommodate these large mesh sizes, techniques are employed to process meshes *in parallel*, on multiple processors simultaneously. *Splatter* is based on MPI [24], *Message Passing Interface* — a distributed memory parallelization framework, widely used for scientific applications. In MPI, processors have their own address space, and network communication is used to exchange values between processors.

Each processor is assigned a range of global nodes for which it is responsible. The region of the mesh consisting of these assigned nodes and the mesh entities that contain them is called the processor's *partition*. Some entities contain nodes belonging to multiple processors. These entities exist along partition boundaries, and will appear in multiple partitions.

Splatter allows the dynamic modification of partitions. Node ownership can freely change, and entities transparently move between processors to appear in the required partitions. Because node ownership is based on global node ids, and processors are assigned

ranges of these nodes, general mesh migration between processors occurs in response to global node renumbering and/or global node distribution range updates. Parallel partitioning as well as load balancing (Chapter 5) are applications of a general renumber/redistribute algorithm (Section 4.3).

3.4 Framework Organization

In *Splatter*, mesh entities are grouped into indices – an abstract term that can safely be thought of as sequential collections (lists) of mesh entities. Different kinds of indices are appropriate for different mesh entities, and these distinctions will be clarified in the forthcoming chapter on implementation (Section 4.2.2).

Regardless of the index type, each index can be associated with arbitrary user data. Data is *attached* to indices using a *Splatter* construct called a data proxy. The data itself remains fully accessible to user code.

When the framework needs to make topological changes to the mesh, or otherwise requires that mesh entities be transferred between processors, the internal changes determined by the application are applied indirectly, to the data proxy, which in turn applies the changes to the user data.

This organization is depicted in Figure 3.5.

The philosophical implications of the proxy concept are significant and reflect strongly on the overall design of *Splatter*. Existing mesh frameworks, as discussed in Section 2.2 expect a lot from their users – they store the mesh in proprietary internal data structures, and expect applications to use their APIs to access mesh entities. Application scientists only need to use *Splatter* calls when manipulating the global mesh.

Furthermore, research [25] indicates an interesting phenomenon present in the field of scientific computing that impacts the adoption of new frameworks, even when the introduced capabilities would be highly desirable: application scientists with practical programming experience are often hesitant to integrate code from outside sources. Through the use of

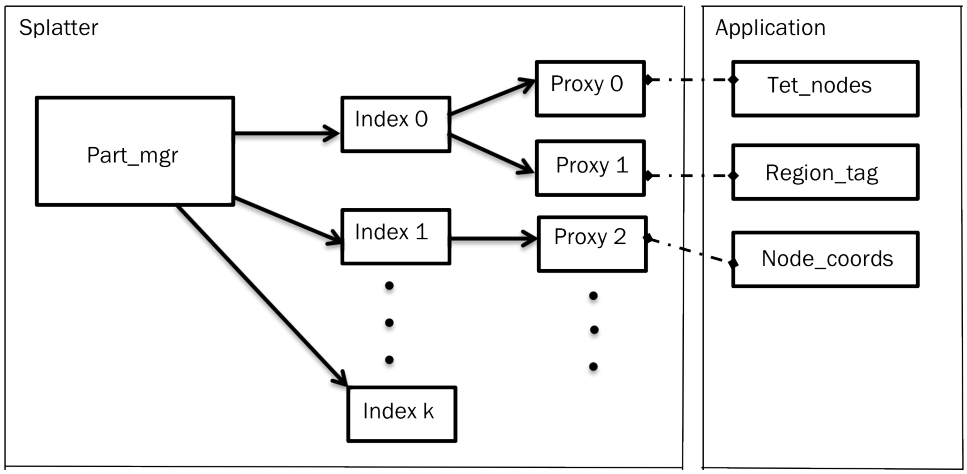

Figure 3.5 Overview of the Splatter architecture

data proxies, *Splatter* encourages iterative integration into existing codes, involving minimal code modification, that may enable more users to feel comfortable adopting the framework.

In a typical *Splatter* application, there is a section of code that does the initial setup: creating the indices of mesh entities managed by the framework, attaching the appropriate data structures (via proxy) to these indices, and potentially registering hooks for when topological changes occur (see Section 5.3). Elsewhere, simple framework calls that perform basic tasks like synchronizing phantom data and saving restart files, as well as more complicated refinement and load-balancing calls, are isolated sections of code — not necessarily impacting any user algorithms.

3.5 Queries

Splatter provides an embedded query syntax that allows the user to process meshes in parallel in an abstract, highly expressive way. This query syntax, along with the user-definable entity schema, completes the interpretation of the mesh as a database.

These queries serve as a replacement for the concept of *iterators* used in other mesh frameworks, and allow the user to traverse, process and extract features of the mesh.

Queries are used for tasks like the following:

1. traversing the complete set of volume entities, perhaps to populate a linear system, as used in PDE-based simulation codes
2. creating a new index containing only the volume entities along a region interface
3. traversing an entity's neighbors
4. iterating over edges that need to be refined, and updating their *upwards adjacent* neighbors (Section 5.3)

Queries are chains of individual *query modules* operating on a stream of mesh entities. These modules can roughly be classified as follows:

1. **sources**: introduce new mesh entities into the stream
2. **filters**: selectively process incoming entities and optionally pass entities to the next module in the chain
3. **terminators**: process incoming entities and terminate
4. **conditions**: process incoming entities and evaluate as *true* or *false* for use in appropriate filters and terminators

This is a non-strict tetrachotomy in that some modules could be classified as both *filters* and *sources*, and it does not adequately provide for the roles of *utility* modules – *technically* filters that provide meta-behaviors such as entity stream routing.

Regardless, it is an extremely flexible system, as these modules can be chained in any syntactically legal order. It is fairly trivial to develop new, application-specific query modules to invoke custom behaviors on the stream.

It should be noted that any particular query functionality is an add-on to the core *Splatter* components. In other words, these query modules are written entirely in terms of public API calls. The significance of this is that alternate query syntaxes and functionality can seamlessly coexist with the existing query functionality.

The true power of this query component is twofold:

1. It enables the succinct expression of complex algorithms, leading to a higher degree of correctness and ease of use.

2. The resulting executable is highly optimized.

3.5.1 An Example

Listing 3.1 contains a typical query. It builds a *result set* containing tetrahedral entities adjacent to a boundary.

```
for_each("boundary") >>=
faces_in(&m,"volume") >>=
save_temp(results) >>=
end()
```

Listing 3.1 A typical *Splatter* query

Recall that a query represents a stream of mesh entities, and contains query modules as classified above.

1. `for_each` is a *source*, introducing new entities into the stream. Every query must begin with a source. In this particular example, `for_each("boundary")` indicates that all entities part of the index named "boundary" should be introduced into the stream.

2. `faces_in` is a *filter*. In this example, the input is the stream of boundary entities as described above. The output is a stream of the entities in the "volume" index that contain those boundaries. Note: the `&m` is a *Splatter* handle, required for efficiency purposes.

3. `save_temp` is another *filter*. It provides a mean for capturing entities in the stream, and is one way of collecting results from a query. The presence of "temp" in the name

indicates that these results should be considered temporary. It would cease to remain valid after structural changes to the mesh.

4. `end` is a *terminator*. In this case it merely indicates the end of the query, and is required.

3.5.2 Standard Query Modules

Table 3.1 documents all standard query modules in the current implementation. Syntactically, queries may contain these in any order, though the first module must be the only source, and the last module must be the only terminator. Certain filters in the table (`skim`, `sort`) refer to a *subquery*. In those cases, any legal sequence of query modules may be a parameter to the filter module. These queries must start with a filter or terminator — not a source. Custom query modules are easy to write, and are described in Section 4.4.1.

Table 3.1 Standard query modules

Name	Parameters	Functionality
Sources		
`for_each`	index, range(*optional*)	stream all entities from index
`all_faces`	index, range(*optional*)	synonym for `for_each`
`one_face`	index, entity id	stream a single entity from index
Filters		
`skim`	condition(*optional*), subquery	if true, copy stream to subquery
`where`	condition	pass entities through when condition is true
`subfaces`	*none*	convert incoming entities to all subentities defined by adjacency
`unique`	optimized(*optional*)	only pass unique entities to output
`count`	int variable	store count of entities passed through in variable
`dump`	ostream variable(*optional*)	print all entities to stream
`flag`	container, value	set `container[entity]` to value
`save_ids`	container	store entity id in container
`save_temp`	temp_index	store entity in temp_index
`save_temp_uniq`	temp_index	store entity in temp_index if it is not already there
`faces_in`	handle[1], index	faces in index containing input stream
`store`	*special*	
Terminators		
`end`	*none*	end the query

Continued on next page

Table 3.1 Standard query modules

Name	**Parameters**	**Functionality**
`split`	subquery1, subquery2, subquery3(*optional*)	route stream to all subqueries
`sort`	*special*	see discussion in Section 4.2.3.3
Conditions		
`flag`	container, value	true when `container[entity]==value`
`owns`	*none*	true when entity is owned on local rank
`phantom`	*none*	true when entity has a non-local node
`needs`	*none*	true when entity has a local node
`negate`	condition	true iff condition is false
`faces_in`	handle, index	returns true if index contains input stream
Other		
`fetch`	*special*	

[1]pointer to core `part_mgr`

3.5.3 Adjacency Queries

To support adjacency search, the standard query modules `subfaces` and `faces_in` are of most use. In terms of the nomenclature introduced in Section 3.2.1.1, `subfaces` replaces incoming entities M_i^d with the downwards adjacent entities $M_i^d\{M^{d-1}\}$.

`faces_in`, when used as a *filter*, relates incoming mesh entities to a user-defined index, I, (e.g., `>>=faces_in(&m,"tets")>>=` is looking in the index named "tets"). Effectively, this module replaces incoming M_i^d with $M_i^d I^x$ in the stream [2] where $x \geq d$ and depends entirely on the nature of I.

Listing 3.2 demonstrates these concepts, determining boundary faces adjacent to an incoming region entity stream.

```
<input stream> >>=
subfaces() >>=                  // yields surface entities
faces_in(&m, "bnds") >>=        // replace with bnds containing subfaces
<output stream>
```

Listing 3.2 Determining boundary faces adjacent to an incoming region entity stream

`faces_in`, when used as a *condition*, does not replace the input stream; it is used to test whether an entity is contained by any index contents. When used with `where`, this acts as a filter of valid entities.

Listing 3.3 demonstrates this usage, generating 2D components from region entities and filtering out those that exist in a "bnds" index, and then removing duplicates. The resulting query stream returns all 2D faces that exist between two region entities.

[2] the contents of I of dimension x

```
{query} >>=                        // incoming region entities
subfaces() >>=                     // constituent face entities

// filter out the ones in a "bnds" index
where(negate(faces_in(&m, "bnds")) >>=

unique() >>=                       // no duplicate faces
{query}
```

Listing 3.3 Filtering boundary entities from a query stream

3.5.4 Result Sets

To facilitate integration with larger algorithms, queries may use `save_temp` or `save_ids` to capture mesh entities from the query stream for further processing. Furthermore, result sets may be shared explicitly with neighboring ranks based on arbitrary user criteria, rather than the default data sharing model imposed by *Splatter* for normal indices. The `sort` query module mentioned in Table 3.1 is used for this purpose, and examples of its use will be seen in Section 5.3.

As with any new language, making best use of *Splatter* queries is a matter of experience. It is very easy to write a query that uses excessive memory and takes excessive time due to redundant processing. It is the hope that as the framework matures and attracts users, static query analyzers may be developed to help quantify the amount of work done by a query and lead to smarter queries.

The software implementation of the query modules, however, is capable of generating extremely optimized machine code. These queries are written entirely in C++, using C++ templates and a technique often referred to as *template metaprogramming*. Through this technique arbitrary user queries are transformed, via standard C++ compilers, into highly optimizable inline function calls.

3.6 Summary of Design Principles

This chapter has provided an overview of the driving design principles behind the *Splatter* framework, and attempted to motivate the implementation details provided in Chapter 4.

In summary, this framework provides extreme user-defined flexibility in terms of what kinds of mesh entities and associated data are managed. It allows the user to process the mesh in terms of high-performance, abstract queries. It integrates with application code in a nonintrustive way.

It is believed that full integration with *Splatter* from the outset of a development project can greatly accelerate the timeline for deliverable software, without imposing ubiquitous, cryptic API calls and design patterns. Users comfortable with the framework can freely involve the use of *Splatter* queries in their algorithms, and benefit from the high-performance and high expressibility they offer. It is hoped that *Splatter*'s extensible query system will foster experimentation and applications in problem domains heretofore unconsidered.

CHAPTER 4
IMPLEMENTATION

4.1 Overview

This chapter discusses the implementation of key data structures and algorithms in the *Splatter* framework. For the full API documentation, see Appendix C. All code is in standards-compliant, portable C++.

4.2 Framework Architecture

As described in Chapter 3, *Splatter* maintains a collection of mesh entity indices, with data *attached* to them. These indices are distributed across the set of MPI processors involved in the mesh. In this section, we discuss the architectural and algorithmic facets of this arrangement, specifically the implementation of the `splatter::index` and `splatter::data_proxy` classes, as well as the global mesh handle and organizational centerpiece: the `splatter::part_mgr`.

4.2.1 The Local Partition Manager

The `splatter::part_mgr` (*partition manager*) class defines the main access point for all *Splatter* functionality. Each process will instantiate one of these objects, per mesh, and it contains all information regarding the local partition. Additionally, `part_mgr` objects act as a handle on the global mesh, and can be used collectively — assuming all distributed `part_mgr`'s agree — to make global changes to the mesh involving node creation and entity distribution (see Section 4.3).

As a central component in the framework, the `part_mgr` contains several members with system-wide significance:

1. an instance of `parallel_ctx`, which encapsulates details of the parallel environment containing the mesh

2. an instance of `ownerdb`, which is the authority on node and entity ownership throughout the framework

3. a collection of `entity_cfg` objects, which define all allowed mesh entity types for the current mesh (see the discussion of adjacency in Section 3.2.1.1).

```
splatter::part_mgr mgr(topo::num_std_entities, topo::std_entities);
mgr.init(splatter::parallel_ctx(my_custom_communicator));
```

Listing 4.1 One way to configure a new `part_mgr`

Several options for constructing and initializing a new `part_mgr` exist – refer to the class documentation in Appendix C. Listing 4.1 illustrates one way to construct and configure a new `part_mgr` to use the framework's standard topological entities on a custom MPI communicator.

4.2.1.1 The Parallel Context

In MPI terminology, a *Splatter* mesh is bound to an entire MPI communicator. Every process participating in this communicator holds a stake in mesh ownership, and participates in mesh operations. Applications may freely create any number of communicators, allowing the existence of multiple meshes simultaneously managed by different sets of processors. Also, a single communicator may own multiple meshes simultaneously, simply by creating multiple `part_mgr` instances.

The `splatter::parallel_ctx` class is an encapsulation of an MPI communicator with a variety of convenience methods used in parallel communication. The `parallel_ctx`

member of `part_mgr` is *the* authority parallel context for the associated mesh, and can be accessed using `part_mgr::pctx()`.

Listing 4.2 contains the salient details of the `parallel_ctx` class's public interface.

```
class parallel_ctx
{
public:
    /*! number of ranks assigned to this MPI communicator */
    int np() const;

    /*! this process's rank on this MPI communicator */
    int rank() const;

    /*!  this MPI communicator */
    MPI_Comm& comm();

    /*! wrapper around MPI_Reduce, using op as reduction function */
    template <typename T> T reduce(T in, MPI_Op op) const;

    /*! wrapper around broadcast, selecting configured root as source of
     *  information */
    template <typename T> void
    broadcast(std::vector<T>& vec, int root=-1) const;
};
```

Listing 4.2 Main functionality of `parallel_ctx`

4.2.1.2 Entity Distribution

In MPI, each processor in a communicator is assigned a unique integer called a *rank* between 0 and NP, where NP is the number of processors in the communicator.

These ranks are used during network communication to identify the endpoints of an MPI data exchange operation. Additionally, many MPI programs are written in a SIMD-style[1], with all processes executing the same code. These ranks are used within application

[1]Single Instruction Multiple Data

code to customize the behavior on each processor; often this simply involves using the rank to choose the data on which to operate.

This pattern is followed within *Splatter*. The global set of node ids is broken into NP contiguous ranges, where again NP is the number of processors in the mesh's parallel context. Each processor is assigned one of these ranges based on its rank.

A role of the `part_mgr` is to ensure that each processor agrees on this node distribution. This responsibility is delegated to a subcomponent of type `splatter::ownerdb` (*ownership database*). This singular `ownerdb` within the `part_mgr` houses the official, mesh-wide node distribution, and provides numerous methods for determining the ownership of nodes and entities in general.

Internally this distribution is stored as a single `int[]` array, called `node_dist`, of size $np + 1$ values (once again, np is the number of processors in the mesh's parallel context). The nodes owned by the processor with rank r are the ones identified by global ids in the range $[\text{node\_dist}[r], \text{node\_dist}[r+1])$. During mesh partitioning, all processors agree on the contents of this array.

The set of node ids distributed across all processors are known as *global* node ids, because they are globally unique over the entire mesh. Since a range of these nodes is assigned to each rank, it is desirable to simplify the way that locally owned nodes are referenced on a given processor. To do this, the concept of *local* node ids is introduced. These ids are *not* unique throughout the mesh, but are unique on a given processor. These node ids always start at 0 on each rank, and extend contiguously to provide an id for each locally stored node.

For example, if rank 3 is assigned global node ids 4032 - 8900, then *on rank 3*, local node 0 maps to global node 4032 and global node 8900 maps to local node 4868.

The node ownership concept corresponds directly with the concept of *partition* introduced in Section 3.3. A processor's partition is defined as the set of nodes assigned to the processor and any mesh entities that contain them.

Due to the presence of mesh entities on partition boundaries, partitions will contain entities that in turn contain nodes owned by other processors.

Application algorithms typically require information about all nodes present in every mesh entity on the local partition. *Splatter* identifies these non-local nodes as remote dependencies, creates local *phantom* nodes linked to them and provides the ability to periodically refresh attached data from their remote counterparts (Section 4.2.3).

These phantom nodes are assigned local ids and are indistinguishable from locally owned nodes. For many algorithms, no special care — other than the periodic synchronization of data — is needed.

The `ownerdb` is responsible for providing all local-to-global node mappings (and vice versa), for all local nodes, including these local phantom nodes.

Listing 4.3 contains the full public interface of the `splatter::ownerdb` class. As discussed, it contains methods for determining global node ownership as well as these local-global node mappings. For efficiency, certain lookup methods have variations to use when it is known that no phantom information is present.

Two important methods available via the `ownerdb` are:

1. `bool owns(int g)`, which returns true if global node `g` is owned by the current rank, and false otherwise

2. `int owner(int g)`, which returns the rank of the processor that owns global node `g`.

The result of `owns` can be determined in $\mathcal{O}(1)$ (constant) time because it requires a simple test to see if `g` is in the global node range for the local rank.

The result of `owner(int g)` can be determined in $\mathcal{O}(\log(np))$ time, where np is the number of processors. This is done using the binary search algorithm seen in Algorithm 4.1.

In cases where authority over a multi-node mesh entity must be exerted, a mesh entity's owner is considered to be the owner of the smallest global node id appearing in the entity. An open question is whether or not this particular entity ownership strategy

```
class ownerdb
{
public:
    bool owns(int g);           // do I own global node g?
    int  owner(int g);          // what rank owns global node g

    int  high();                // what is my highest global id
    int  low();                 // what is my lowest global id

    int  g2l(int g);            // global->local
                                // (assumes owns(g))

    int  l2g(int l);            // local->global
                                // (assumes l is valid local)

    int  g2l_p(int g);          // global->local
                                // (using phantom lookup)

    int  l2g_p(int g);          // local->global
                                // (considering phantoms as local)

    int  nlocal();              // number of locally owned nodes
    int  nglobal();             // number of global nodes in mesh
    int  nphantom();            // number of phantom nodes
    int  nlocal_p();            // number of local nodes + nphantom
}
```

Listing 4.3 The public interface of `splatter::ownerdb`

input : g – global id
input : node_dist – node distribution array of size np
low ← 0
high ← np
repeat
  mid ← (low + high) / 2)
  if $g <$ *node_dist[mid]* **then**
    high ← mid
  else if $g >=$*node_dist*$[mid + 1]$ **then**
    low ← mid+1
  else
    return mid
  end
until *low >= high*
illegal node
return -1

Algorithm 4.1 Determining global node owner using node_dist

introduces unacceptable biases in parallel algorithms, perhaps causing lower ranks to receive a disproportionate amount of work. No such problems have been detected during the development of this framework.

4.2.1.3 Entity Type Configuration

Splatter is designed to allow user-defined mesh entity types. These types are specified by the user as instances of `splatter::entity_cfg`.

Each `entity_cfg` specifies:

1. a string identifying the name of the entity (used for debugging)
2. the number of nodes required for an entity of this type (usually the number of vertices in the polytope)
3. the dimension, dim, of this geometric shape (currently unused by the implementation)
4. the number of constituent subentities (called subfaces) of dimension $dim-1$ (downward adjacencies)
5. for each subentity
   (a) the type of the subentity
   (b) the tuple of entity nodes (positions) needed to generate the subentity

Listing 4.4 contains the C++ definition of the `entity_cfg` data type, as well as the creation of the `entity_cfg` instance corresponding to quadrilaterals in *Splatter*'s standard entity list.

`entity_cfg` objects always appear in contiguous arrays of compatible, mutually-defined objects. One of these arrays is always *active* meaning it is the current set of allowed entity types. With this in mind, entity types are referred to universally by the integer

```
struct entity_cfg
{
    std::string name;
    int dim;          /* dimension */
    int nnodes;       /* number of nodes */
    int numsubfaces;  /* number of sub entities */

    struct
    {
        int subface_type;
        int subface_nodes[SPLATT_MAX_SUB_ENTTITY];
    } subfaces[SPLATT_MAX_SUB_ENTTITY];
};

/* definition of topo::QUAD */
entity_cfg[] std_entities =
{
    ...

    { "quad", 2, 4, 4,
      {
          { topo::EDGE, { 0, 1 } },
          { topo::EDGE, { 1, 2 } },
          { topo::EDGE, { 2, 3 } },
          { topo::EDGE, { 3, 0 } }
      }
    }

    ...
};
```

Listing 4.4 Definition of `entity_cfg` and configuration of `topo::QUAD` instance

position of the corresponding `entity_cfg` value in the containing array. `topo::EDGE`, in Listing 4.4, is an integer equal to the position of a similar `entity_cfg` definition for edges.

Notice how `topo::QUAD` is defined recursively in terms of `topo::EDGE`. Read the subface list as follows:

1. Edge 1 contains nodes 0 and 1 from the quad.
2. Edge 2 contains nodes 1 and 2 from the quad.
3. Edge 3 contains nodes 2 and 3 from the quad.
4. Edge 4 contains nodes 3 and 0 from the quad.

These recursive entity type definitions are processed by the framework to automatically generate adjacency information between entity types. For example, based on the definitions of adjacency introduced in Section 3.2.1.1, A quadrilateral is *upwards* adjacent to its constituent edges, because it is defined in terms of them. Any other shapes that are *upwards* adjacent to edges are potential *neighbors* of the original quadrilateral.

These two specific values — `topo::QUAD` and `topo::EDGE` — are provided by the standard entity list that comes with the framework. `topo::std_entities` contains internally consistent definitions of node, edge, triangle, quadrilateral, tetrahedron and pyramid. An example of configuring a new `part_mgr` with these standard entities was seen in Listing 4.1.

4.2.1.4 Other Functionality

The `part_mgr` contains other functionality and data for facilitating the processing of the mesh. Most importantly, the `part_mgr` houses all entity indices — this is the subject of Section 4.2.2. Important global algorithms accessed through the `part_mgr` are the subject of Section 4.3.

Here is a list of minor functions that are worthy of note. Documentation and demonstrations of these are available in the Appendix.

1. the capability to save and load *restart files* containing arbitrary user data, allowing applications to restore previous states of processing

2. modification hooks, which allow the user to specify arbitrary code that should be executed when topological changes occur in the mesh. These are used extensively in the refinement and coarsening demonstrations in Chapter 5.

4.2.2 Mesh Entity Indices

Splatter indices are sequential collections of mesh entities. They are constructed and accessed via the `part_mgr`.

Examples of typical indices include:

1. the collection of all local nodes
2. the collection of all surface boundary entities
3. the collection of all internal volume entities

`splatter::index` is a virtual superclass, currently extended by two different concrete index types, and infinitely extensible to support future applications. The superclass provides two main sets of functionality:

1. the ability to associate index contents with arbitrary user data – a process referred to as *attaching data*. This is explored in detail in the forthcoming section on *data proxies* (Section 4.2.3).
2. the ability to quickly iterate over index contents

These indices are associated with data, such that every entity in an index has a corresponding entry in the associated data.

Examples of typical user data include:

1. spatial coordinates of individual nodes (associated with all local nodes)
2. boundary condition tags (associated with all boundary entities)
3. volume condition tags (associated with all volume entities)

4.2.2.1 Explicit Indices

The primary subclass of `splatter::index` is `splatter::explicit_index`. A single `explicit_index` contains entities of a single entity type, and thus may be considered a *homogeneous* collection of mesh entities. It is called an *explicit* index, because it *explicitly* contains all the nodes of all contained entities.

These indices are distributed across all mesh processors. When mesh movement occurs (Section 4.2.3) explicit entities that are directed to move to other processors will appear in the corresponding explicit index on the target processor.

In addition to storing these entities, the `explicit_index` maintains a search structure, called a `fast_index`, for quick node → entity lookups, meaning "given a node id, quickly determine all entities that contain it."

Listing 4.5 illustrates the creation of an `explicit_index` for storing entities of type `topo::TET`. The variable `mgr` is the `part_mgr` object deigned to contain the new TET index. This particular index is given the name "tets".

```
std::vector<int> tet_nodes;

// missing code that loads tet_nodes with values
...

idx = mgr.add_explicit_index("tets", topo::TET, tet_nodes,
                             SPLATT_STEAL_DATA |
                             SPLATT_PHANTOM_DEPS |
                             SPLATT_FAST_INDEX);
```

Listing 4.5 Creating a new `explicit_index`

`topo::TET` is the standard entity type for tetrahedra. This type is defined to require 4 nodes per entity, providing an obvious interpretation of the `int`s contained in the `tet_nodes` `vector`: specifically, there are 4 nodes per entity in the index, and thus `tet_nodes.size()/4` total tetrahedra.

The final argument to `add_explicit_index` is a bitwise-OR'd collection of special *Splatter* flags, which configure the behavior of this index, as well as the `part_mgr`'s relationship with the index. These particular flags may be interpreted as follows:

1. `SPLATT_STEAL_DATA` – the index should remove the contents of `tet_nodes` and store the node values internally. This is appropriate for use in a subroutine that sets up the initial `part_mgr`, since `tet_nodes` may now be safely discarded. Without this flag, the `vector` continues to store all node data, and the `explicit_index` processes the node contents using a dynamically created proxy.

2. `SPLATT_PHANTOM_DEPS` – nodes included in `tet_nodes` that are not owned locally should introduce phantom node dependencies into the `part_mgr` (see the discussion of `ownerdb` in Section 4.2.1.2).

3. `SPLATT_FAST_INDEX` – the framework should optimize internal data structures, potentially at the expense of extra memory, to facilitate the quick lookup of entities containing a given node.

Generally, a statement such as that seen in Listing 4.5 would occur in an isolated "setup" method. Once created, *Splatter* assumes full responsibility for the contents and parallel distribution of the entities contained in `tet_nodes`.

4.2.2.2 Entity Search

Section 3.2.1.1 introduced the concept of *circular adjacency*. `explicit_index` objects provide the implementation of this concept, and are used during queries to extract topological mesh features.

To satisfy adjacency searches, particularly those accomplished using queries, any index must provide the ability to quickly search for entities containing a given node.

The logic behind this was presented in Section 3.2.1.1, particularly in the discussion of *circular adjacency* data structures: because we can use the `entity_cfg` structure to generate all subfaces of an entity, if we can find the *parent* entity quickly using an `explicit_index`'s search functionality, then we can find all subface entries (perhaps in other indices) that contain a given node.

The framework class that contains circular adjacency information is called `splatter::fast_index`. Instances of this class provide, effectively, a mapping from global ids to sets of entities containing them.

Listing 4.6 contains the public interface of the `fast_index` data structure. The primary use of these objects is through the overloaded `operator[](int g)`, which returns

a `splatter::intset` – a custom data structure representing a unique set of integers – populated with the entities containing `g`.

```
class fast_index
{
public:
    fast_index();

    // specify the range of "local" ids
    fast_index(int low, int high); // range of 'local' ids

    // update the range of ints we consider "local"
    status update(int low, int high) ;

    intset& operator[](int g);


    const intset& operator[](int g) const;

    void clear();


    /** erase unused global structures **/
    void purge();
};
```

Listing 4.6 The public interface of `splatter::fast_index`

Originally, this functionality was implemented using standard C++ containers: `std::map<int, std::set<int> >`, that is a `map` from `int` to `set`s of `int`s. This implementation was problematic in two ways:

1. The memory usage was extremely high. Internally, both the `map` and `set` implementations involved a large number of individual allocations, resulting in a ridiculously large *page table size* (a standard memory management data structure used in various operating systems) for a moderately sized mesh.

2. The standard `set` container does not provide any mathematical set operations, which are very useful for adjacency searches.

The `fast_index` implementation cut memory usage by more than half, and uses a custom data structure for integer sets (`splatter::intset`) that provides a fast set intersection operation (again, useful for adjacency queries).

More significantly, map lookups with `fast_index` are much faster than those with the standard `map` container: Since `map` is tree-based, the time taken for any lookup is $\mathcal{O}(\log N_{local})$, where N_{local} is the number of local nodes. With the `fast_index`, most node lookups are $\mathcal{O}(1)$.

To achieve the $\mathcal{O}(1)$ performance, the following observation was made: because of the nature of *Splatter* partitioning, *most* node lookups are for those nodes owned by the local rank. To optimize the `fast_index` for local node ranks, the index uses a `std::vector<intset>` to store local node results, and a `std::map<int,intset>` to store results for all other nodes (mostly phantom nodes). An overwhelming majority of node lookups occur with local nodes, so most of the actual lookups take place in constant time using the contiguous `vector`. Looking up non-local nodes operates in $\mathcal{O}(\log N_{\text{phantom}})$, where $N_{phantom}$ is the number of non-local nodes present in the index. If a partitioning is *good* (Section 5.2), $N_{phantom}$ is *much* smaller than N_{local}.

Another optimization that takes place within the `fast_index` concerns the rather dynamic nature of partitioning over the course of some *Splatter* applications. Generally speaking the range of *locally owned* node ids could change rapidly as entities are shifted subtly between partitions, such as during load-balancing. To compensate for this, the range of nodes considered *local* is decoupled from the actual range defined by the partitioning.

When a mesh redistribution results in a different range of assigned global nodes, the `fast_index` members of the various indices are *updated* to use the new range. Within the code for `fast_index::update`, heuristics are evaluated to determine whether or not to physically update the internal index components (that is reassigning the node values handled by the `vector` versus the `map`).

Generally, the algorithm decides that the index should be updated if the discrepancy between the old and new node ranges would result in *too many* local node ids falling outside the range handled by the `fast_index`'s `vector`. *Too many* is defined by a constant provided when the framework is compiled (`SPLATT_FI_RATIO_THRESH`).

The end result is a very fast search structure that responds intelligently to mesh updates. It choose to perform these relatively expensive index updates *only when* the benefit outweighs the expense.

The `splatter::intset` objects used in the `fast_index` allow very fast *set intersection* operations. To check for the existence of a specific entity in an index, simply intersect the sets returned from the `fast_index` for each of the nodes in the entity. If the resulting set intersection contains any values, then the desired entity is present, specifically it exists at the locations contained in the resulting intersection. Each individual search takes $\mathcal{O}(1)$ time, on average, and the time for set intersections is linear in the size of the set ($\mathcal{O}(|\text{intset}|)$), also known as "the average number of entities per node". The resulting performance analysis is $\mathcal{O}(\text{width} \times \text{average}|\text{intset}|)$.

For typical polytopal complexes, this equates to a constant.

This entity lookup algorithm was presented in Algorithm 3.1.

The search data structure and entity lookup algorithm work perfectly for *correct* mesh entities in a true polytopal complex. The flexibility inherent in *Splatter*'s entity type configuration, however, allows for mesh entities that are not correct (see an example in Section 5.3.4.1).

Entities used in atypical applications could contain duplicate node numbers. Applying the lookup algorithm in such a case could result in false positives: matches that do not truly contain the desired search term.

Atypical algorithms could also define non-polytopal entities, with subentities that do not appear as faces or edges in the shape (perhaps an internal edge connecting nonadjacent nodes). With the current algorithm, searching for such an edge could return false positives — entities that contain the searched-for edge nodes, but without the actual connecting edge.

These are generally experimental applications, and there are ways to account for this possibility. One option is to define a new `index` type with a more appropriate search structure.

Alternately, the `fast_index` can be used to narrow down search possibilities, but the results from the search must be verified with a second-stage test. That is the approach taken in Section 5.3.4.1.

As suggested above, alternatives to *circular adjacency* exist and are desirable for certain applications. This current work is completed entirely using this adjacency structure, but new indices could be created (as proper, separate extensions of `splatter::index` that provided alternate search functionality).

One reasonable approach would be to incorporate a half-edge data structure [26, 27] — this search mechanism maintains explicit *upwards adjacency* information unlike the current implementation, and thus could prove more efficient for certain kinds of queries.

4.2.2.3 Implicit Indices

`implicit_index` objects represent a sequence of entities defined *implicitly* by a range of values, rather than by explicitly enumerated entity tuples. These are used exclusively for representing collections of nodes.

The `part_mgr` maintains a single `implicit_index` exclusively for representing the sequence of local nodes. These nodes are stored implicitly in the index as the range of global values assigned to the local processor, augmented by the set of phantom nodes stored in the `ownerdb`.

Implicit indices offer no search mechanism or other capabilities. (Searching, in this case, is handled entirely by the `ownerdb`.) The primary role of these indices in the framework is to allow the attachment of user data to the nodes themselves, without having to store the node ids.

The real value of `implicit_index` will become clear during the discussion of data proxies in Section 4.2.3.

4.2.2.4 Attaching Data

Recall that an index is a sequence of entities. Attaching data involves taking a similar sequence of data elements and pairing it up with the index. The sequences must be of the same length, and the values match up position-by-position. That is, the 0^{th} entity owns the 0^{th} data element; the 50^{th} entity owns the 50^{th} element, etc.

A data sequence *attached* in this way becomes managed by the `part_mgr`. As mesh entities are moved between processors, or reordered locally, the attached data elements follow.

4.2.2.5 Working with Indices

In many programming environments maximum flexibility and high-performance are at odds with each other. Throughout *Splatter*, however, a consistent implementation pattern is followed that enables both flexibility and high performance.

This implementation pattern is relevant to the discussion on indices because working with mesh indices is a fundamental operation in the framework. It must be fast.

Splatter is written entirely in C++, a programming language commonly used in scientific computing. Because this language is compiled directly for a target CPU architecture, it is possible to generate extremely fast programs. A major factor dictating the speed of a compiled C++ application is the quality of the compiler's optimizer.

A general rule of thumb for writing code that is to be processed by an optimizing compiler is "give the compiler as much information to work with as possible". By following this rule, the developer can lean heavily on the expertise and experience of the compiler developer and take advantage of architecture-specific optimizations, making great strides towards having "fast code" with, otherwise, little effort.

Unfortunately, certain aspects of C++ tend to undermine the efficacy of optimizers. Consider, specifically, the concept of polymorphism, from the Greek meaning *many shapes* — a great promise of object oriented programming. The "normal" polymorphic approach is to envision a hierarchy of data types. Algorithms written in terms of the general superclasses

of the hierarchy may be seamlessly applied to instances of the derived superclasses. Specific functionality from the subclasses will *kick in* during the algorithm, causing the code to *change shape* in response to the specific data types involved.

This is a powerful programming model for complex systems, but — at least in C++ — this flexible behavior is only possible when several criteria are met, each of which has a detrimental effect on performance:

1. The general data structure must be a pointer or reference (aka a fancy pointer).
2. The methods called on the general data structure must be declared `virtual`.

The problem with pointers and references is the level of indirection between the variables being operated on, and the data being manipulated. Using pointers violates our first rule of writing optimizable code: the compiler does not have access to information surrounding the actual data being operated on, just the pointers!

Access to these values can not be optimized by the compiler, because the values themselves are not yet defined. They are going to be *pointed at* during runtime.

Virtual methods, introduced by the `virtual` keyword, push this concept even further. These virtual members are identified by the compiler as elements of a class that may have been overridden in a subclass. Because of this possibility, at runtime when such a member is referenced, the objects must be inspected to determine which method implementation is appropriate (not unlike the use of standard function pointers).

Code that calls a virtual function will always require the use of the call stack. Parameters are pushed on to the stack, a subroutine call is invoked with the virtual subroutine code pointer, and when that routine is finished, the stack is popped.

When high performance is required, such as in a tight number-crunching loops, virtual function calls should be avoided.

On the other end of the performance spectrum lies *inline functions*. Functions of this type are entirely accessible — source code and all — to the optimizing compiler. Inline function calls can be replaced by the compiler with the actual code from the function body.

Call stack manipulations are removed from the optimized code, allowing the compiler to further optimize access to that function's local variables.

4.2.2.6 Iterating over Virtual Indices

To optimize access to `virtual` subclasses of `splatter::index`, techniques involving C++ templates are used. These techniques, sometimes referred to as *compile-time polymorphism* are designed to preserve the aspects of inheritance that lend themselves well to the design of complex systems, while still allowing the compiler to highly optimize the code.

We have the general problem of needing to iterate over the contents of an `index`, without using `virtual` accessors to access the mesh contents.

The solution for the case of `index` is representative of the framework philosophy in general: rather than iterate over the index contents, have the index *apply* a functor[2] to itself. This is implemented as a non-virtual templated method `apply` on the `virtual index` superclass.

Listing 4.7 shows the definition of `index::apply`. C++ does not allow virtual templated methods, but this implementation is *spiritually* virtual, in that a call to this method immediately and explicitly translates into a call to `do_apply` on the actual subclass object.

Two downsides exist, but neither is intractable:

1. Adding a new subclass of `splatter::index` requires modification of this `apply` method.

2. When `index::apply` is expanded by the template processor, it in turn expands all subclass versions of `do_apply`, even if they are not called. This would result in a negligible amount of *code bloat* – binary contents in the executable resulting in higher memory requirements.

[2] often called a *function object*

```
template <typename OP, typename IT>
void index::apply(const OP& op, IT start, IT end)
{
    switch(_type)
    {
     case SPLATT_IMPLICIT:
         static_cast<implicit_index*>(this)->
             do_apply<OP,IT,&OP::operator()>(op, start, end);
         break;
     case SPLATT_EXPLICIT:
         static_cast<explicit_index*>(this)->
             do_apply<OP,IT,&OP::operator()>(op, start, end);
         break;

     default:
         ERROR("cannot 'apply+iterators' to index with type " << _type);
    }

    return;
}
```

Listing 4.7 Apply a functor to a range of mesh entities in an index.

`do_apply` is defined for each concrete `index` subclass. The `do_apply` method explicitly iterates over the index contents, applying the functor to each entity. Since `do_apply` is in turn templated in terms of the functor, the code for the functor is available to the optimizing compiler during the compilation of `do_apply`. The end result is a tight loop over the `index` contents calling the body of functor as if it were declared `inline`.

For this to work on any index, functors must be defined in such a way that they can process any mesh entity. This simply means that the functor must accept *general enough* parameters that could represent any mesh entity.

This is formalized through the `splatter::index_op` class (Listing 4.8). Specifically, any functor that extends this class can be used in a call to `index::apply`. Because this is handled by the template processor, there is nothing special about this `index_op`. It merely exists to help the user write a functor class that will work. Attempting to implement the `operator()` method, but failing to take the right parameters, will result in a compile-time error since the abstract `virtual` operator has not been properly provided.

```
class index_op
{
public:
    // idx -- an index
    // eltno -- entity position within index
    // etype -- entity type for this entity
    // nnodes -- number of nodes in this entity
    // nodes -- actual nodes in this entity
    virtual void operator()(splatter::index* idx, int eltno, int etype,
                            int nnodes, int* nodes) const =0;

};
```

Listing 4.8 Expected functionality of a functor used in `apply`

To reiterate: the `index_op` class is unknown in the `apply` method, so no attempt at calling this `virtual` operator will ever be attempted. The native code in the actual subclass will be expanded via template, and in turn highly optimized.

4.2.3 Data Proxies

Data proxies are objects that serve as mediators between parallel *Splatter* algorithms and proxied local data structures. They are designed to implement specific parallel data movement *protocols* by translating them into operations on arbitrary data structures.

They are of particular use when dealing with `index` objects and their attached data. Protocols are initiated with a data structure describing details of the corresponding data movement. That data structure can be used to invoke a protocol on the data proxy managing an index's entity nodes as well as on every proxy managing data that is attached to that index. The end result is the equivalent data movement operations (defined by the proxy) being applied to each individual data sequence (the index itself, as well as each attached collection of data elements). All data is moved *in the same way* and ends up reunited in their new location

As discussed earlier, indices and data elements are related by their positions in the corresponding sequences. Those positional relationships are preserved even though the entities and data may be moving to entirely new processors.

4.2.3.1 Data Movement Protocols

The following data movement protocols are supported:

1. *batch migrate*: for arbitrarily copying data to other processors and deleting local copies
2. *reorder*: for local data reordering, using an *in situ* permutation algorithm
3. *sync*: for copying data to and from remote processors into preordained locations, such as is used for refreshing phantom node data
4. *deliver*: for packaging up data into a payload for other MPI ranks (does not affect the local data).

```
class data_proxy
{
public:

    virtual status batch_migrate(const migrate_args& args)=0;

    virtual data_proxy* deliver(const deliver_args& args)=0;

    virtual status sync(parallel_ctx pctx, const sync_args& args)=0;

    virtual status reorder(const std::vector<int>& localids)=0;
}
```

Listing 4.9 Declaration of data proxy protocols

4.2.3.2 The `batch_migrate` *Protocol*

`batch_migrate` is used for general data movement between processors, supporting `explicit_index` objects. Its primary argument, an instance of `migrate_args` contains the following:

1. a `std::deque<int>` containing data (identified by position) that should be removed from the underlying data structure

2. a `std::vector<std::deque<int> >` containing the data (identified by position) that must be copied to each MPI rank.

Upon the first use of a new `migrate_args`, a coordination phase occurs, in which all involved processors communicate message sizes and initialize communication buffers.

Each invocation of the `batch_migrate` protocol packages up the proxied data and communicates it with the other relevant processors. Incoming data overwrites those data elements marked for deletion.

Since the protocol is applied in the same way to the mesh entities and all attached data, the position integrity of the index and data is preserved.

The cost of applying this protocol is analyzed in this way:

1. construction of the `migrate_args` is part of a larger algorithm, potentially user-defined, but generally consists of a process that looks at each mesh entity and classifies it based on where it should be moved. This is a $\mathcal{O}$(number of local entities) calculation.

2. Preparing the arguments involves an all-to-all MPI call, $\mathcal{O}(np)$, where np is the number of processors.

3. The actual migration of data occurs once for each related data proxy, and takes time proportional to the amount of data moved: $\mathcal{O}$(number of data proxies $\times$ amount of data).

The end result is, on average, as good as can be expected: $\mathcal{O}(np +$ amount of data to move).

4.2.3.3 The `deliver` Protocol

`deliver` is a special protocol used in conjunction with query result sets. A special query syntax, based on the `sort` query module, allows for the easy construction of `deliver`

protocol arguments. Functionally speaking it is similar to `batch_migrate`, but has no facility for deleting arguments. Unlike `batch_migrate` which is designed for application in distributed indices, `deliver` is used to copy a set of values into a remote query result set.

4.2.3.4 The `reorder` Protocol

`reorder` is used to rearrange local data, based on a `std::vector<int>` which containing new local destinations for each rank. This is used primarily for the renumbering of an `implicit_index`. It operates by using an *in situ* permutation algorithm that runs in $\mathcal{O}(n)$ time with minimal memory overhead.

Algorithm 4.2 contains the algorithm implemented by the `reorder` protocol.

4.2.3.5 The `sync` Protocol

`sync` exists to support proxies on node data. As described in Section 4.2.1.2, phantom nodes are assigned local ids, and the `ownerdb` maintains a mapping from local ids to their remote counterpart. These mappings are used to construct a `sync_args` object for parameterizing the `sync` protocol.

The `sync` protocol is invoked on a single `data_proxy` attached to node data, and results in refreshed values for the attached data associated with all phantom nodes.

4.2.3.6 Standard Data Proxy Implementation

The primary implementation of `data_proxy` is a templated subclass that operates on behalf of arbitrary `std::vector` objects. This class exists because the `std::vector` is a highly efficient container providing nice manipulation semantics on top of a standard, cache-friendly, contiguous array.

The implementation of `data_proxy` protocols in terms of `std::vector` methods was very straightforward and performs excellently. Furthermore, in the interest of appealing to scientists with existing codes, the `std::vector` is syntactically compatible with plain C

```
input  : data – array of arbitrary user data
input  : localids – new local positions for each element in data
declare: prev – array of integer positions, initialized to -1

for n ← 0 ... data.size do
    find beginning of cycle
    if prev[n] == -1 and localids[n]!=n then
        p ← localids[n]
        prev[p] ← n;
        until the end of the cycle..
        while p ≠ n do
            prev[localids[p]] ← p
            p ← localids[p]
        end

        tmp ← data[n]

        copy backwards through cycle
        while prev[p] ≠ n do
            data[p] ← data[prev[p]]
            p ← prev[p]
        end

        data[localids[n]] ← tmp
    end
end
```

Algorithm 4.2 Reorder local data using *in situ* permutation

arrays. The process of incorporating *Splatter* with existing scientific codes based on arrays is discussed in Section 5.4;

4.2.3.7 Arbitrary Data Types in Parallel

Throughout the framework, parallel operations on arbitrary user data are performed. From the framework's perspective, the user data types are unknown, and thus are implemented as template arguments, `T`, throughout the framework API. (See, for example, the definition of `parallel_ctx::reduce` in Listing 4.2).

To convert between an unknown type `T` and a valid `MPI_Type`, the framework uses a templated helper subroutine called `get_mpi_type`.

```
template <typename T> MPI_Datatype get_mpi_type(T ex)
{
    ERROR("using base get_mpi_type(ex)!"); return MPI_BYTE;
}

template <typename T> MPI_Datatype get_mpi_type()
{
    ERROR("using base get_mpi_type()!"); return MPI_BYTE;
}
```

Listing 4.10 The base templated `get_mpi_type` methods

Listing 4.10 contains the base definition of `get_mpi_type`. Note that this base version of the subroutine does not function: the `ERROR` macro triggers an `MPI_Abort` — early termination of the executing code (`return MPI_BYTE` exists here only to satisfy the requirements of the compiler that the subroutine return an `MPI_Datatype`).

C++ *template specifications* providing concrete values for `T` are used to override this base template at compile time, resulting in inline functions that return the specific appropriate `MPI_Datatype` values as needed. Two versions of the method exist depending on whether access to the raw type data is convenient or an example value must be used to trigger the proper template application.

Template specifications for the most common data types are seen in Listing 4.11. These enable the templated `parallel_ctx` methods, as well methods in `data_proxy` to operate on arbitrary data.

Even more interesting is the example seen in Listing 4.12 in which a new `MPI_Datatype` is constructed when first called. This new type represents `count` contiguous `T` values — perfect for when an array of values needs to be considered a unit. The templated

```
template<>
inline MPI_Datatype get_mpi_type<double>(double ex)
{
    return MPI_DOUBLE;
}

template<>
inline  MPI_Datatype get_mpi_type<double>()
{
    return MPI_DOUBLE;
}

template<>
inline  MPI_Datatype get_mpi_type<int>(int ex)
{
    return MPI_INT;
}

template<>
inline  MPI_Datatype get_mpi_type<int>()
{
    return MPI_INT;
}

template<>
inline  MPI_Datatype
get_mpi_type<std::complex<double> >(std::complex<double> ex)
{
    return MPI_COMPLEX;
}

template<>
inline  MPI_Datatype get_mpi_type<std::complex<double> >()
{
    return MPI_COMPLEX;
}
```

Listing 4.11 Template specifications for returning appropriate `MPI_Datatype` values

function maintains a static map of generated `MPI_Datatype` values, allowing unique types for different sized arrays.

This pattern of providing a template specification of `get_mpi_type` that creates a new complex type on the fly can easily be followed, by framework users, to provide automatic system-wide parallel support for custom data types, for example using `MPI_Type_create_struct`.

```
template<typename T>
MPI_Datatype get_mpi_type(T example, int count)
{
    static std::map<int,MPI_Datatype*> types;

    std::map<int,MPI_Datatype*>::iterator match = types.find(count);

    if (match == types.end())
    {
        LLOG(2, "creating new mpi_type for count");
        MPI_Datatype& newtype = new_mpi_type();
        MPI_Type_contiguous(count, get_mpi_type(example), &newtype);
        MPI_Type_commit(&newtype);
        types[count]=&newtype;
        return newtype;
    }
    else
    {
        return *(match->second);
    }
}
```

Listing 4.12 A more complicated template specification for more a complex data type

With `get_mpi_type` thus defined, a user can write an algorithm that transfers arbitrary user data using MPI. This can be seen in Listing 4.13.

```
template <typename T>
T add_up(T in)
{
    T rval;
    MPI_Allreduce(&in, &rval, 1, get_mpi_type<T>(),
                  MPI_SUM, MPI_COMM_WORLD);
    return rval;
}
```

Listing 4.13 Using `get_mpi_type`

4.2.3.8 Reflection

Data proxies support minimal *reflection* capabilities for application that must copy unknown user data between proxies.

Use the `data_proxy::type_copy()` method to return a new `data_proxy` capable of holding the same type of data as the original.

Use `data_proxy::store_raw()` to copy data from a position in a source proxy into a local position. This assumes that the source proxy is *type compatible*, such as one created using `type_copy()`.

4.3 Mesh Renumbering and Redistribution

In Section 3.3, the partition was defined as the collection of all locally owned nodes and the mesh entities containing them. This partitioning can be modified in two ways:

1. The ranges of nodes owned by each process can be modified (redistribution)
2. Nodes can be reassigned new global ids (renumbering)

This section covers the implementation of *Splatter*'s `part_mgr::renumber` method, which provides a single access point for both global renumbering and redistribution.

`renumber()` takes a `vector<int>` containing the new global ids for every locally owned node, and a `vector<int>` corresponding to the desired new node distribution (Section 4.2.1.2).

```
status renumber(const std::vector<int>& newgids,
                const std::vector<int>& dist,
                std::list<temp_index*> externals=std::list<temp_index*>());
```

Listing 4.14 Method signature for `part_mgr::renumber`

The algorithm proceeds by renumbering each index separately. The individual `index` subclasses define `renumber` in such a way that entity nodes are replaced with their replacements, and the entities are moved to the processors that require them, based on the new node distribution. The entity movement is always conceived in terms of `data_proxy` protocols, meaning that entities and their associated data appear in their new locations together.

Indices that are automatically updated when a `renumber` occurs are called *tracked indices*. All indices created via the `part_mgr` are automatically tracked. The third argument to `renumber` in Listing 4.14 is a manual list of `temp_index*` objects that should be temporarily tracked. `temp_index` is used to store query result sets, so this temporary tracking allows result sets to be renumbered along with the rest of the mesh.

`temp_index` instances are not distributed across processors, so no automatic entity movement is performed, just node renumbering.

Algorithm 4.3 illustrates the renumbering process for `explicit_index`. This algorithm is operates in $\mathcal{O}$(number of local entities) time.

To renumber an implicit index, the `batch_migrate` protocol is applied such that the data of every local node is routed to its new owner. Additionally, this protocol is applied to the vector of new global ids. The result is that node data is associated with its new global id on the destination processor!

After the batch migration, every processor has a complete, yet out-of-order collection of all local nodes associated with their new global ids (and no extraneous nodes). These global ids are translated to new local node ids using the `ownerdb`'s `g2l()` (global to local id) method, and the results are used as input to the `renumber` protocol.

```
declare: bma – new batch_migrate args

for each entity in index do
    determine new numbering for all nodes

    for each new owner o of entity nodes do
        register with bma that entity should copy to o
    end

    if entity is no longer needed locally then
        register with bma that entity should be deleted locally
    end
end

invoke batch_migrate protocol on entity index and all attached data
```

Algorithm 4.3 Global renumber and redistribute for explicit indices

The initial batch migration takes $\mathcal{O}(N_{local})$ time on each processor, as does the *in situ* permutation that takes place during the `renumber` protocol. Implicit index renumbering is linear in the size of the local partition ($\mathcal{O}(N_{local})$).

4.3.1 Hooks

To facilitate integration with user code, applications may register *renumber hooks* – arbitrary code that should be executed whenever the global `renumber` method is invoked. These methods can be used, for example, to rebuild local connectivity data structures that are outside the domain of *Splatter*, such as linear system connectivities.

4.3.2 Related Algorithms

Renumbering can also be used to introduce new nodes into the mesh, such as during mesh refinement. Simply create a new node distribution that accounts for extra unused

nodes on each processor, and optionally calculate new global ids for each node so that they will remain in the range of nodes owned by the local processor.

Removing unused nodes can be accomplished using an obscure feature of the `renumber` method. Create a new node distribution array of a smaller size and assign each unneeded node the new global id of -1. The implementation will remove those nodes, and their associated data, completely.

4.4 Query Implementation

Splatter's embedded query syntax is based on standard C++ templates. A technique called *template metaprogramming* leverages the template processor to create new objects based on syntactic structures appearing in the code.

Specifically, consider a simple query that prints out all of the entities in an index, Figure 4.15.

```
for_each("tets") >>= dump() >>= end()
```

Listing 4.15 Query that prints out all index contents

This chain of query modules is translated, by the template processor, into a temporary class object that iterates over the contents of "tets" and evaluates the `dump` module on each one (the dump module just prints out the nodes of the incoming entity).

The `>>=` operator, typically used for *auto right shift* — a fairly obscure bitwise operator — has been overloaded with a template, to take two arbitrary values as operands, as seen in Listing 4.16. This operator returns an instance of a special `join` class that unites the two operands in a special way.

```
template
< typename T1,
  typename T2 >
join<T1,T2> operator >>=(T1 x, T2 y)
{
    return join<T1,T2>(x,y);
}


template <typename T1, typename T2>
class join : public query_op
{
public:
    join (T1 x, T2 y) : x(x), y(y) {}

    void operator()() const
    {
        x(y);
    }

private:
    T1 x;
    T2 y;
};
```

Listing 4.16 Overloaded `>>=` operator and `join` class

At compile time, when the template processor sees the first `>>=` operator in Listing 4.15, it determines that for the operator use to be syntactically valid, then it must be syntactically valid to build a `join<for_each,dump>` class.

Looking in the `join` template definition, when creating a `join<for_each,dump>` class the instance variable `x` is of type `for_each` and `y` is of type `dump`. These two instance variables come together in `join::operator()` with the single statement that unlocks the mystery of the entire query syntax: `x(y)`.

`x` must have an `operator()` that is compatible with `y`. In fact, this operator, defined for `for_each` objects, simply iterates over all entities in the index specified at construction time (in this case "tets"), and for each entity applies the functor `y`, which happens to be a `dump`.

Every query module follows this pattern. The parentheses operator is overloaded to call and/or be called by other query modules. These calls are chained together via `join` classes gluing the objects together. The end result has the feel of a stream of entities flowing through the chained modules.

Technically, the `>>=` operator is right associative (which is part of the reason this operator was chosen for queries), so this process would have happened first in analyzing the relationship between `dump` and `end`.

`dump` is a *filter* module (Section 3.5) that takes incoming entities. This means that the `operator()` is designed to take arguments that reflect a single, general mesh entity. After it does its processing (prints the entity nodes to standard output) it calls the next one in the `join`-chain with the same entity. Here, that next module is the special `end` module, a *terminator* that simply receives incoming mesh entities and stops.

The actual temporary class generated by the template processor is of type `join<for_each, join<dump,end> >`. This process repeats for the entire sequence of chained query modules.

After the template processor creates this temporary class, the compiler is invoked to translate it into machine code. The optimizer has access to all of the class code for every piece involved: all of the `operator()` calls may be inlined, and further optimized, until the machine code is indistinguishable from a hand-written loop that explicitly processes the mesh in the prescribed way.

4.4.1 Custom Query Modules

Recall the general classification of query modules from (Section 3.5): *sources*, *filters*, *terminators* and *conditions*. The distinction between these module types is entirely dependent on how the `operator()` receives and passes along mesh indices.

These calls must be defined in terms of general enough parameters that any mesh entity in any index can be passed. In this current implementation, those parameters are as follows:

1. `splatter::index* idx` – the containing index
2. `int eltno` – entity position within that index
3. `int etype` – the mesh entity type for this entity
4. `int nnodes` – number of nodes in the mesh entity type (redundant, for efficiency)
5. `int* nodes` – the actual node ids of the entity

To facilitate the development of new query modules, these required arguments are bundled together in standard C macros, based on the desired role of the module in a query. These macro definitions are listed in Listing 4.17.

```
#define SQ_SOURCE(out)
#define SQ_TERM(idx, eltno, etype, nnodes, nodes)
#define SQ_FILTER(idx, eltno, etype, nnodes, nodes, out)
#define SQ_COND(idx, eltno, etype, nnodes, nodes)
```

Listing 4.17 Macros to facilitate custom query development

These macros are used to define all the standard query modules, and may be used in user code as well.

Consider, for example, the `split` module, a terminator that routes incoming entities to 2 output modules. In Figure 4.18, the use of `SQ_TERM` is demonstrated to implement `split` functionality.

Queries represent a particularly novel contribution to the field of mesh management frameworks. Future work will focus on adding capabilities and additional performance improvements to this feature of *Splatter*.

```
template <typename T1, typename T2>
class split_obj : public query_op
{
public:
    split_obj(T1 x, T2 y) : x(x), y(y) {}

    // call 'x' and 'y' on entity
    SQ_TERM(idx, eltno, etype, nnodes, nodes)
    {
        x(idx,eltno,etype,nnodes,nodes);
        y(idx,eltno,etype,nnodes,nodes);
    }

private:
    T1 x;
    T2 y;
}
```

Listing 4.18 `split` routes incoming entities to two subqueries

CHAPTER 5
APPLICATIONS

5.1 Overview

This chapter discusses several applications of the *Splatter* framework, demonstrating the appropriateness and usability of the framework for advanced tasks.

5.2 Initial Partitioning

Splatter is designed to manage the computational mesh throughout an application run. In the beginning of a typical run, the mesh must be loaded on all parallel processors before the application can proceed. This framework offers a variety of functions for the *initial load* of mesh data. In this section, these various functions are explained and applied to real mesh data.

5.2.1 Support for Mesh Formats

There are generally two different approaches to the initialization of a *Splatter* instance, depending on the nature of the data being loaded into the system. When a new mesh is processed for the first time, it must be manually loaded into the framework for processing. Subsequent runs can employ the framework's standard *save* and *restart* capability that allows an application to resume execution from an arbitrary state reached during an earlier application execution.

The process of manually loading a new mesh into *Splatter* for the first time depends entirely on the format of the mesh generation output process. Generally speaking, any format that supplies the mesh in terms of numbered polytopes (see Section 3.2) is usable with *Splatter*.

The core data structures do not provide general or specific IO functionality other than the *save/restart* capability mentioned earlier. Any reader for a specific mesh file format must be written to use the official, exposed public interfaces of *Splatter* data structures.

That said, subroutines have been written for reading monolithic *Star-CD* formatted ASCII mesh files, in parallel, in a way that is compatible with *Splatter* indices. (By *monolithic*, it is meant that the mesh data is provided in large files without any regard for parallel distribution.) These subroutines are used throughout the applications discussed in this chapter, and are illustrated in the sample code provided in Appendix A.

These `starcd::read` routines proceed by having each processor open the mesh files simultaneously, jump to a position in the middle defined by the processors rank and the total file size, and reading in an assigned part of the file. This process is known as *striping*, and results in the mesh data being arbitrarily distributed over the involved processors.

These routines are not particularly efficient in the way that they access the disk. Explicitly using multiple simultaneous file pointers to read a file from a parallel file system can, anecdotally, lead to some unsatisfactory performance. Specifics of this performance degradation will be seen in the analysis to follow.

These performance issues could be resolved with a new mesh file reader based on MPI IO – a standard mechanism for reading files in parallel that avoids these observed performance issues. MPI IO, however, does not work well with ASCII based files and thus would not work with Star-CD mesh files, which were chosen for other practical reasons.

Recall that any mesh-format specific file reader will exist entirely outside the framework. The relative performance of mesh loading mechanisms that are not part of fundamental *Splatter* functionality are simply not a concern in this present work.

What is a concern, however, is that this striping process results in poor quality partitions (Section 3.3) in which the partition boundaries, shared between processors, are large compared to the computational volume of each partition. Figure 5.1 illustrates a 2D slice of a 3D mesh loaded on 4 processors using this striping technique. In this figure, each

processor is assigned a color. Each processor does own the same number of nodes, but they are distributed throughout the domain haphazardly.

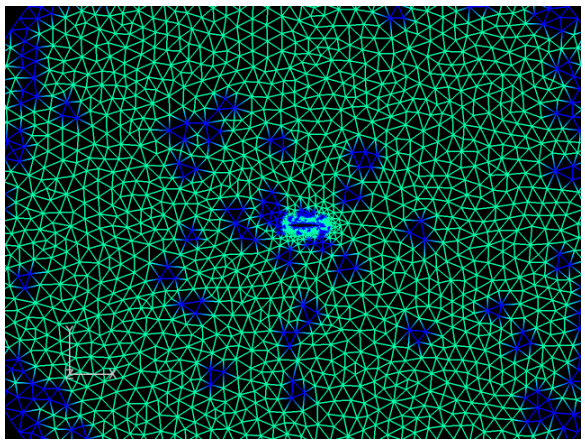

Figure 5.1 Striping alone results in a poor partitioning

The subject of partition quality has been researched extensively (see, for example [28],[29], [30]). The most desirable partitioning maximizes local computation and minimizes inter-process communication — that is, minimizes the number of phantom nodes and shared entities along partition boundaries, while maximizing the amount of local calculations. This is an optimal configuration due to the relatively high cost of communication when compared to computation [31].

In *Splatter*, a striping load must be followed by an explicit parallel partitioning phase. The framework provides flexibility for working with a number of existing mesh partitioning software libraries, including Zoltan [32] and – perhaps the *de facto* standard – ParMETIS [33], a parallel implementation of the highly regarded mesh and graph partitioning library, METIS [28]. Basically any algorithm that can assign each local node a target partition can be used with this framework. In general, these tools work by taking a global mesh connectivity as input, and assigning the nodes an appropriate partition.

The METIS family of software libraries implement a *multilevel k-way partitioning* algorithm that has shown to generate excellent partition quality and very fast performance.

This is the particular partitioning software and algorithm chosen for this test of the *Splatter* framework.

To apply the metis algorithms, particularly the `ParMETIS_V3_PartKway` subroutine, the mesh must be transformed into its *graph dual* or node-to-node connectivity in the commonly used compressed row storage (CRS) form. This is accomplished this using the `index_op` defined in Listing 5.1.

The output of the metis algorithm is a partition assignment (an `int`) for each node. Using Algorithm 5.1, these partition assignments are translated into new global ids for each node (note: not all ids will change). Furthermore, the number of nodes per partition leads directly to the new node distribution array (Section 4.2.1.2). These new ids and new distribution are input for *Splatter*'s global `renumber/redistribute` algorithm (Section 4.3) which results in all tracked indices and associated data moving to new locations on new processors, as appropriate.

5.2.2 Partition Quality Analysis

Figure 5.2 shows the mesh from Figure 5.1 following an application of a ParMETIS-based renumbering algorithm. Visually it is clear that the nodes assigned to each rank are no longer haphazardly distributed throughout the mesh.

To quantify the improved partition quality, a striped load followed by a ParMETIS-based renumbering was tested on two fairly large simplicial (tets, triangles) meshes. These meshes are listed in Table 5.1.

Table 5.2 and Table 5.3 contain, for the `single_block` and `eight_block` meshes respectively, the minimum and maximum partition sizes (in terms of entity counts) before and after an explicit ParMETIS-based renumbering, with a variety of processor counts (np).

The partitioning process attempts to group nodes into equal-sized partitions, while minimizing the dependencies between partitions and maximizing the number of entities on a partition (referred to as the *computational volume*, earlier). From these tables, it can be

```
class CSRBuilder
{
public:
    // this is applied to each index entity
    void operator()(splatter::index* idx, int item, int etype,
                    int width, int* nodes) const
    {
        for (int n1=0; n1<width; n1++)
        {
            if (_odb.owns(nodes[n1]))
            {
                int l = _odb.g2l(nodes[n1]);
                // remember connectivity between n1 and n2, for nodes
                // owned locally
                for (int n2=0; n2<width; n2++)
                {
                    if (n1 != n2)
                    {
                        _hash[l].insert(nodes[n2]);
                    }
                } } } }

    void getCSR(std::vector<int>& ia, std::vector<int>& ja)
    {
        // preallocate ia,ja
        ...

        // load ia,ja with connectivity
        for (unsigned int n=0; n<_hash.size(); n++)
        {
            ia.push_back(ja.size());
            for (unsigned int v=0; v<_hash[n].size(); v++)
            {
                ja.push_back(_hash[n][v]);
            }
        }
        ia.push_back(ja.size());
    }

private:
    std::vector<intset>& _hash;
    const ownerdb& _odb;
}
```

Listing 5.1 The `index_op` used to build the compressed row storage of a mesh

```
input  : parts – array of partition ids for each local node
input  : nparts – the number of partitions
input  : np – number of MPI processors involved in decomp
input  : rank – local MPI rank

output : new_globals contains the new global id for each local node
output : new_dist nparts+1 integers reflecting new node distribution

declare: part_sizes_local – array of size npart
declare: part_sizes_global – array of size nparts × np
declare: part_offsets – array of size nparts
declare: tally – an integer

for for each part in parts do
    part_sizes_local[part]++
end

get all part sizes from all involved ranks
MPI_Allgather( part_sizes_local → part_sizes_global )
part_sizes_local ← 0...0

tally ← 0
for p ← 0 ... nparts − 1 do
    for r ← 0 ... np − 1 do
        if r = rank then
            part_offsets[p] ← tally
            tally ← tally + part_sizes_global[r × nparts + p]
            part_sizes_local[p] ← part_sizes_local[p] + part_sizes_global[r × nparts+p]
        end
    end
end
tally ← 0
for n ← 0 ... nparts do
    new_dist[n] ← tally
    tally ← tally + part_sizes_local[n]
    new_dist[nparts] ← tally
end
for n ← 0 ... new_globals.size − 1 do
    new_globals[n] ← part_offsets[parts[n]]
    part_offsets[parts[n]] ← + part_offsets[parts[n]]
end
```

Algorithm 5.1 Calculating new global ids from partition output

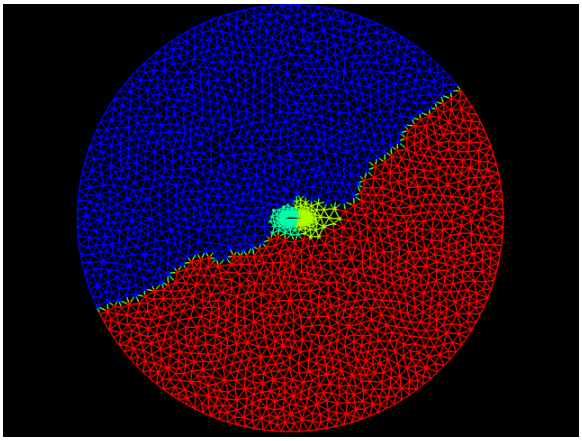

Figure 5.2 A mesh with a high-quality partition

Table 5.1 Sizes of the simplicial meshes used for testing

mesh name	node count	tetrahedron count	triangle count
`single_block`	6798161	40025670	692596
`eight_block`	54385252	320205360	5540768

seen that the initial *striped* partitioning had a very even node distribution, in terms of *node count per partition*.

The tet distribution, however, varied widely because the count included entities along the partition boundaries, which were arbitrary (see Figure 5.1). The ParMETIS process assigned a *less even* node distribution, but the resulting entity balance was much better.

The higher-dimensional entity distribution has a profound impact on run-time performance for typical PDE-based simulation applications. The dependencies between nodes yield non-zeros in the representative linear system, and thus directly contribute to the size of the problem and the amount of work to be done per rank. In short, the tetrahedron count tends to reflect the work per process more than the node count.

Additionally, the tetrahedra counts are high because of the excessive size of the partition boundary, in this striped case. All partitions contain a high number of entities containing non-local nodes because no efforts were made to minimize the inter-partition

Table 5.2 Mesh entity counts affected by ParMETIS on the `single_block` mesh.

np	state	nodes,min	nodes,max	tets,min	tets, max	bnds,min	bnds,max
16	pre	424885	424886	2966550	5209003	0	692596
	post	394447	432097	2351357	2688059	4228	63012
32	pre	212442	212443	1318682	3198179	0	455006
	post	198964	216139	1234627	1358827	0	38665
64	pre	106221	106222	667539	1909987	0	277692
	post	97718	107934	589655	695072	0	22226
128	pre	53110	53111	338209	1069662	0	171944
	post	47429	53960	294088	353588	0	14261
256	pre	26555	26556	208240	568869	0	104093
	post	22790	26964	148151	180416	0	8791

dependencies. This leads to a much higher communication demand when updating data for phantom nodes.

Furthermore, the imbalance noted in post-ParMETIS results is configurable in the particular parmetis call used for partitioning. Specifically, an *imbalance threshold* parameterizes the partitioning algorithm. Here the suggested default value of 1.05 is used. This can be decreased, allowing less imbalance. Here we define the imbalance of X as $\frac{X_{max}-X_{min}}{2\times(X_{max}+X_{min})}$, the magnitude of the range of values divided by the average value.

A last item to consider, these resulting partition sizes — particularly for the `eight_block` case — are extremely high for typical simulation applications. A mesh containing 320 million tetrahedra would typically be processed with a much larger computational system, containing potentially thousands of processors. In such a case, the individual partition sizes would be much smaller, and ParMETIS's allowed imbalance threshold would result in count discrepancies of a much smaller magnitude.

5.2.3 Partitioning Scalability Analysis

Measuring the time it takes to apply the initial partitioning is very interesting in that it suggests certain trends regarding the performance of *Splatter*'s internal data structures and parallel communication algorithms.

Table 5.3 Mesh entity counts affected by ParMETIS on the `eight_block` mesh

np	state	nodes,min	nodes,max	tets,min	tets, max	bnds,min	bnds,max
16	pre	3399078	3399079	21092647	24367788	0	5540768
	post	3349934	3448224	19839594	20418826	332053	362608
32	pre	1699539	1699540	10219866	14060663	0	3445374
	post	1645069	1726669	9768743	10312563	147892	184532
64	pre	849769	849770	5134431	8878685	0	1771018
	post	791311	864666	4731051	5212246	59965	107857
128	pre	424884	424885	2550270	5606273	0	904880
	post	365327	432399	2202616	2677209	4412	66611
256	pre	212442	212443	1310011	3296484	0	498548
	post	184512	216327	1117522	1372091	0	40676
384	pre	141628	141632	860661	2429394	0	343352
	post	117012	144041	717083	921297	0	30986
512	pre	106221	106225	665911	1925250	0	289514
	post	94144	108116	579290	694960	0	24968

Considering only the `eight_block` case, Table 5.4 shows the time (in seconds) taken to load and partition the `eight_block` mesh in a variety of parallel contexts. In this table, the rows have the following meaning:

1. Striped read: time to load the Star-CD file
2. Build conn: time to generate connectivity input for ParMETIS process
3. ParMETIS call: time for partitioning algorithm to complete
4. Mesh movement: time to renumber in response to partitioning
5. Phantom sync: time to collect phantom nodes from entities on partition boundaries and their associated data

Table 5.4 highlights two problem areas for the current implementation:

1. The striped mesh reader really does perform poorly as the number of processors increases (as per the discussion in Section 5.2.1).
2. ParMETIS does not scale with the number of processors!

Table 5.4 Timing of initial striped load and partitioning on the `eight_block` mesh

np	16	32	64	128	256	384	512
Striped read	660.1	401.4	246.1	273.3	396.4	601.7	595.5
Build conn	12.3	6.7	4.2	2.4	1.3	.75	.61
ParMETIS call	29.8	22.4	61.5	81.5	172.0	272.2	308.1
Mesh movement	67.6	37.8	23.4	18.5	9.9	7.5	6.2
Phantom sync	.34	.23	.17	.16	.06	.09	.08
Total time	770.14	468.53	335.37	375.86	579.66	882.24	910.49
Total - read	110.04	67.13	89.27	102.56	183.26	280.54	314.99

Problem 1 from this list will be ignored for now. As discussed above, it is not germane to the performance of *Splatter*, as the load subroutine exists outside the framework. A proper MPI IO implementation is practically guaranteed to alleviate this issue.

Problem 2 is more of an issue, and it's no surprise — ParMETIS is known to perform better when using fewer processors. Note: this does not mean it must create fewer partitions. This issue is addressed in the next section, Section 5.2.4.

An interesting thing about the problem of initial partitioning is this: the amount of work required to go from a striped file to quality partitions is highly variable. It depends entirely on the attributes of the monolithic mesh files. As the number of partitions increase, the amount of data that is "probably on the wrong partition" increases, and the global work required to put everything where it belongs increases.

A simple thought experiment backs this up: if we desired a single partition, then all data from the monolithic mesh file would be on the desired partition, immediately upon reading the mesh in serial — requiring no extra work. Alternately, if we had a processor for every node in the mesh, then, then every tetrahedral entity must be copied to at least 3 other partitions.

With this degree of arbitrariness, the general performance reflected in Table 5.4 — other than the problems identified above — is highly acceptable.

The amount of time it takes to prepare the connectivity for ParMETIS scales reasonably (approximately 60% scalability over the tested range of processor counts, for a component of the algorithm that is inherently fast).

Similarly, the amount of work required to physically move all mesh data in response to the partitioning (labeled *Mesh movement* in the table) scales reasonably, especially considering the thought experiment posed above.

5.2.4 ParMETIS Improvements

It is clear from these results that ParMETIS should not be invoked on all processors involved in the mesh for the initial partitioning. We have implemented a method that allows fewer partitions to take part in the ParMETIS execution. Unfortunately, the current implementation requires the processors involved in ParMETIS to physically store the entire mesh while doing so. This will be improved in the next revision of the code.

Specifically, we create an MPI group, with a corresponding communicator, that handles all of the initial load and distribution of the mesh. This will be replaced in the future with a variation that allows the mesh data to be read in parallel on the entire complement of processors, while using a restricted (smaller) set of processors just for the ParMETIS algorithms.

Regardless, the current system does allow us to reason about the performance aspects of the ParMETIS algorithms using fewer processors for the partitioning.

Table 5.5 illustrates timing information for these alternate configurations. Particularly, the `eight_block` mesh is always loaded and partitioned on 32 processors, farming the partitioned mesh out to the entire complement of processors.

Table 5.5 Load and partitioning on fewer processors

np	32	64	128	256	384
num loaders	32	32	32	32	48
Parmetis	22.4	32.3	32.3	33.5	33.1
Mesh movement	37.8	45	45	45.8	32.2

In summary, improved scalability of the initial partitioning can be achieved with a rewrite of the ParMETIS-related code. A current implementation can be invoked on a more desirable number of processors, but the entire mesh and all data must fit in the memory allocated to the executing processes, along with the communication buffers needed to distribute the mesh entities. Because of the memory size of the communication buffers, the cases for 384 and 512 processors above could not be loaded on 32 processors. The 384 processor case was partitioned on 48 processors, and the 512 processor case was not attempted.

Splatter is capable of performing the initial parallel load and distribution of a mesh suitably well. Certain optimizations must be made to improve the performance of the system *in general*.

Note: In the introduction to this section, a system for *saving* and *restarting* the processing of a mesh was mentioned. These mechanisms are used later in this chapter, and do not suffer from either the IO bottlenecks or ParMETIS limitations discussed here. Specifically, the partitioned mesh can be saved in this format as soon as partitioning is complete. In this mode, partitioning can be treated as an offline process, meaning that future runs need not concern themselves with the initial distribution of the mesh at all.

5.3 Adaptive Refinement and Coarsening

Splatter supports topological changes to mesh entities over the course of an application run. Such changes impact internal framework data structures, namely the node $\rightarrow$ entity maps (Section 4.2.2.2) and phantom node mappings (Section 4.2.2.3). Because of this, it is imperative that these changes either occur via framework API calls that maintain internal data structure integrity, or are explicitly followed by calls that rebuild these internal structures (such as `part_mgr::augment_nodes` and `explicit_index::build_hash` — see Appendix C).

In this section, we describe an advanced application of the *Splatter* framework that allows for the dynamic refinement and de-refinement of simplicial meshes in response to user-defined refinement functions.

In a simulation application, these user-defined refinement functions would typically correspond to interpretations of solution data, such as high gradients in solution variables. Indeed, this particular application will be shown in Section 5.4.

Here, we focus on the mechanics of refinement and de-refinement, using an entirely artificial refinement function. As such, we can explore the behaviors of extreme mesh modification, without the overhead of actual simulation code numerics.

The test function we use is a sphere around an arbitrary coordinate point in the mesh. Entities that intersect this sphere will be *fully refined*. As the experiments progress, this sphere will move through the mesh, refining entities along its path, introducing many additional nodes and mesh entities. In response to these dynamic mesh modifications, the mesh partitioning will be explicitly updated to rebalance partition sizes.

Finally, de-refinement is allowed. In this case, refined entities that are *too far away* from the refinement sphere are restored to their original configuration. This is a highly complicated algorithm that makes novel use of *Splatter*'s flexible index structure and query syntax.

5.3.1 Refinement of Simplicial Entities

The purpose of mesh refinement techniques is to dynamically add extra nodes to a mesh as an application progresses. In the regions of the mesh where these nodes are added, mesh entities are replaced with new, smaller entities defined in terms of the additional nodes. With more nodes in these regions, a higher resolution solution can be determined, hence this technique is useful in regions of the mesh where high solution activity is detected.

Certain application techniques, based on particular equation discretization schemes, include mathematical specifications for estimating the numerical discretization error of an

ongoing solution, throughout the mesh [34]. Such a scheme could be used to decide where mesh refinement should occur, minimizing the detected error.

Splatter's refinement capabilities are separate from any such mathematical motivation. With this framework, the user — based on an arbitrary criterion — merely indicates which mesh entities should be broken into smaller pieces.

The algorithm used by the framework is flexible and extensible, but the current implementation is limited to simplicial meshes.

Every tetrahedral entity marked for refinement by the user will eventually be replaced by 8 smaller tetrahedra. A new node is introduced along each edge in the original, and these split edges define the resulting smaller shapes. Figure 5.3 illustrates the full refinement of a single tetrahedron.

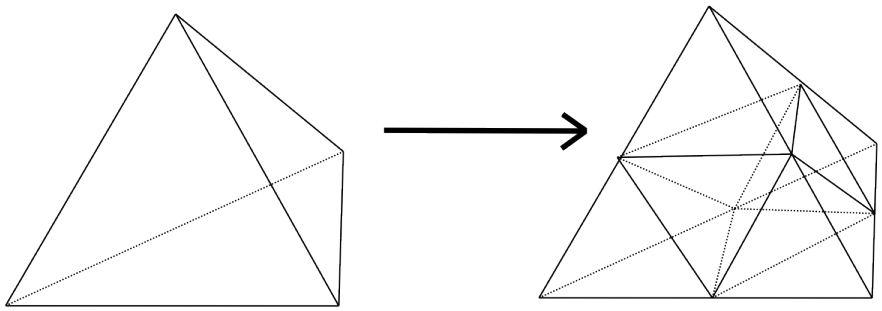

Figure 5.3 Full refinement of a tetrahedron

Replacing these tetrahedra alone will not yield a valid polytopal complex. There must be transition entities acting as a layer between the fully refined and unrefined entities. This concept is demonstrated using triangular entities, for clarity, in Figure 5.4. (Note: some applications may allow *hanging nodes* between adjacent mesh entities — that configuration is supported by *Splatter*, but not used in this demonstration).

Further complicating this refinement algorithm is the fact that it is happening in parallel. Special attention must accompany mesh operations along partition boundaries. All

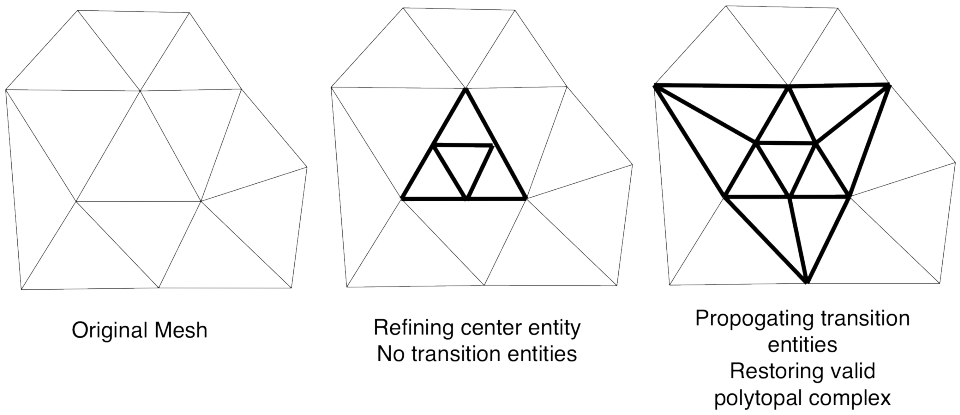

Figure 5.4 The need for transition entities between fully refined and unrefined entities

processors affected by an entity refinement must agree on how to proceed. The framework provides considerable help in designing algorithms that handle these special cases.

5.3.1.1 Mesh Refinement Algorithm

The global mesh refinement algorithm proceeds in 3 stages:

1. Mark all edges that need to be split, propagating through transition entities until the proposed configuration yields a valid complex.

2. Request from the framework enough new global nodes to assign one to each refined edge.

3. Replace all entities containing a marked edge with the appropriate configuration of smaller entities such that all marked edges are split.

Stage 1 of this global algorithm proceeds according to Algorithm 5.2. This algorithm handles the determination of transition entities according to a user-specified entity classification function. It propagates edges marked for refinement until the resulting discretization eliminates all hanging nodes. The end result is a list of all edges that must be refined by the local processor. Note: as mentioned the current implementation is explicitly

for simplicial meshes, but this algorithm is written in terms of general entities subjected to edge-based refinement.

```
input  : refine_edges – set of edges required to split
input  : indices – collection of Splatter indices
output : refine_edges – set of all edges required to split, leaving a valid complex
declare: new_edges – extra edges refined during this iteration

Tell other ranks about refine_edges, on partition boundaries, storing incoming edges
in refine_edges

repeat
    new_edges ← ∅
    foreach entity ent in indices adjacent to any of edges do

        invoke geometry-specific test
        if not TEST(ent, edges) then
            introduce extra edges needed for a valid refinement
            new_edges ← new_edges ∪ ADDITIONAL_EDGES(ent, edges)
        end

        Tell other ranks about new_edges, storing incoming edges in new_edges
        refine_edges ← refine_edges ∪ new_edges
    end
until global(new_edges) ≠ ∅
```

Algorithm 5.2 Propagating edge refinement

The purpose of the entity classification test (corresponding to TEST and ADDITIONAL_EDGES in Algorithm 5.2) is to ensure that when an entity is refined, the resulting replacement entities are valid and are suitable for the application. The current implementation of this TEST, for tetrahedra, requires that no tetrahedron have more than 3 refined edges without being *fully* refined. This is intended to halt the creation of small tetrahedra with high aspect ratios, and to provide a model for future refinement of more

complex entities. The corresponding ADDITIONAL_EDGES implementation marks extra edges for refinement, as needed, to satisfy this TEST.

Since any call to ADDITIONAL_EDGES has the affect of causing the main propagation loop to repeat, the process continues until all parallel ranks agree that all entities will end in a valid state (`global(new_edges)` in the algorithm translates to an `MPI_Reduce` operation). Having too strict a TEST would cause this process to repeat indefinitely, with the result of refining far more entities than would be desirable.

An obvious improvement to this algorithm would be the ability to mark certain edges as *unrefinable*. With this the user would be able to keep the transition entity propagation process from venturing into regions of the mesh where refinement should not occur, for whatever reason.

5.3.1.2 Actual Refinement Implementation

Algorithm 5.2 is written for clarity. Several implementation details require further explanation.

Listing 5.2 contains the actual query used to implement the inner loop of the algorithm — basically a single pass in the transition entity propagation process. It uses a single custom query module to simultaneously provide the TEST and ADDITIONAL_EDGES functionality. The `deliver` protocol (Section 4.2.3.1), as applied to the `temp_index` result set called `new_edges` is used to communicate refinement information with adjacent processors.

Specifically, this query tests every mesh entity that is adjacent to any edge on the list to be refined. The `store/sort` combination routes all newly refined edges to processors impacted by the refinement — specifically, those that share a stake in any affected mesh entity.

The custom `validate_tet` query module is designed to analyze an incoming entity and make sure that the set of edges to be refined will leave that entity in a usable state. This is where custom user criteria for refinement would go. This module introduces new edges

to be refined as appropriate, and these are the edges that are routed to the appropriate neighboring ranks. The source code for `validate_tet` is included in Appendix B.

```
for_each(refine_edges) >>= faces_in(root_idx) >>= unique() >>=
    >>= store(all_needers(true), "tet_needers")
    >>= validate_tet(refine_edges)
    >>= save_temp(new_edges)
    >>= sort(fetch("tet_needers"), face_id(), new_edge_delivery)
```

Listing 5.2 Splatter query for propagating refined edges in parallel

Once this algorithm identifies all edges that must be refined, the `part_mgr` is requested to provide enough new global ids to assign one to each of the refined edges. This `get_more_nodes` algorithm proceeds by calculating a new, larger, node distribution containing an appropriate number of unused nodes on each rank. This new distribution is applied using the standard `renumber/redistribute` (Section 4.3) functionality. The new/unused nodes are associated with each edge-to-refine and these mappings are routed using framework queries and *deliver* to processors in need of the data.

Lastly, a counterpart to the `validate_tet` query module is applied that actually replaces every refined entity with the appropriate smaller entities. Specifically, this module is called `refine_tet` and it has the luxury of assuming that `validate_tet` succeeded in enforcing a valid refinement plan for each tetrahedron. When the application of `refine_tet` is complete, the indices will contain the replacement tetrahedra in place of the originals.

5.3.1.3 Hooks

To further satisfy arbitrary user applications, code may be registered with the framework to execute when new nodes and entities are created. Code thus registered is considered a *hook* (or a callback) and is used typically to copy user-specific data into the replacement entities.

For example, region entities are often associated with volume condition tags. Using a refinement hook, the user can specify that when an entity is replaced during the refinement process, the new smaller entities should inherit the original's volume condition tag.

The presence of these hooks enables seamless integration of the described refinement functionality with user applications.

5.3.2 Refinement Results

To demonstrate this refinement capability, we reintroduce the demonstration described earlier in this section. An analytic refinement function defined as a sphere around a coordinate point is introduced and moved through the mesh over the course of several iterations. All mesh entities intersected by the sphere are fully refined. Transition entities are propagated as needed.

For illustration, Figure 5.5 is a sequence of images generated from this test program, on a smaller 3D mesh of 17016 nodes and 75576 tetrahedra, using 4 processors. Each contributing rank is assigned a color.

Table 5.6 contains results from this test program run on the `eight_block` mesh consisting of 54 million nodes and 320 million tets. The refinement algorithm scales well — very close to linear scaling — but the resulting partition is poorly distributed among the processors. This imbalance leads to poor parallel performance in solve applications (see Section 5.4).

The initial imbalances reported in Table 5.6 are allowed, as they fall within ParMETIS's prescribed imbalance factor. Refining the mesh introduces imbalance, which leads to idle processors. The increased imbalance as the number of processors increases makes sense: the refined mesh entities are consolidated onto fewer, smaller partitions, which get proportionally larger than those containing no refinements.

Considering the relatively low percentage increase in average nodes per partition (.26%), the magnitude of the resulting imbalance is striking.

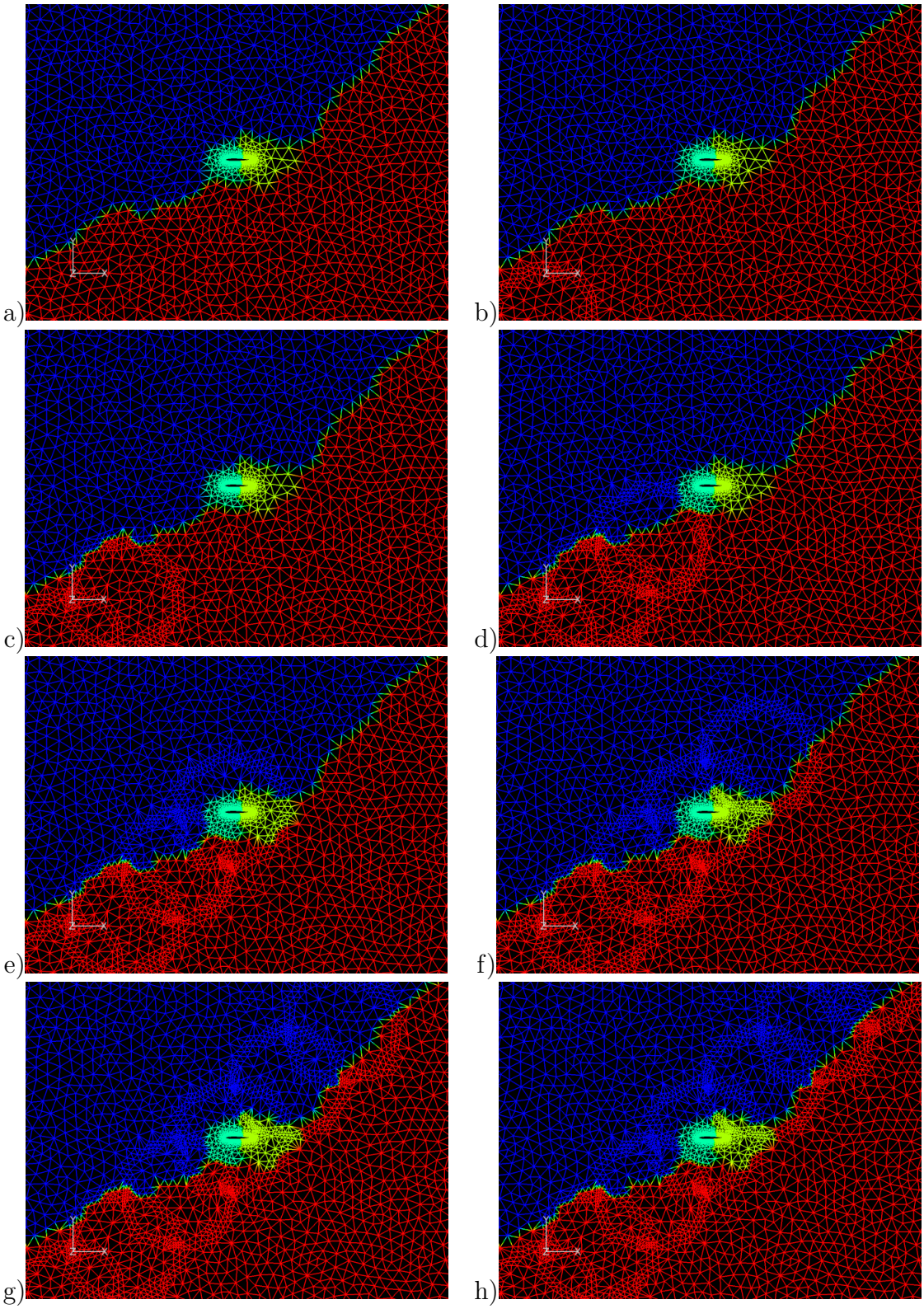

Figure 5.5 Moving the refinement function through the mesh

Table 5.6 Partition imbalance introduced by mesh refinement

np	32	64	128
before refinement			
minimum node count	1683624	820010	385772
maximum node count	1770382	902895	466734
node imbalance	**1.2%**	**2.4%**	**4.7%**
minimum tet count	9768743	4731051	2202616
maximum tet count	10312563	5212246	2677209
tet imbalance	**1.3%**	**2.4%**	**4.8%**
avg. delta node count	4511	2257	1146
avg. partition size increase	.26%	.26%	.26%
refinement time (s)	72.7	52.3	20.8
after refinement			
minimum node count	1683624	820010	385772
maximum node count	1874249	1023201	581509
node imbalance	**2.7%**	**5.5%**	**10.1%**
minimum tet count	9768743	4731051	2202616
maximum tet count	10896768	5915415	3329890
tet imbalance	**2.7%**	**5.6%**	**10.1%**

5.3.3 Load Balancing

As entities are refined, the additional nodes introduced are owned by the rank that owned the refined entity. Left unchecked, this leads to a load imbalance in which certain processors have an excess of work to perform. Processors with less work to do idle, waiting for the busy processors to finish. The result is a poor use of computational resources, and extra time required for the application to complete.

This load imbalance occurs because after refinement, certain processors are responsible for a disproportionate number of mesh entities. To restore the balance of load, the mesh is *adaptively repartitioned*.

Splatter uses ParMETIS routines, again, to calculate new, more balanced partitions. In fact, the previous load-balancing algorithm is used here, with one small but important modification. `ParMETIS_V3_PartKway` is called with an `adaptation` flag, indicating to the

partitioning library that the existing partition was *mostly good* (which is true), it just needs to be adjusted. The use of this flag causes ParMETIS to execute in a mode that does not suffer from the scalability problems discussed in Section 5.2.4.

Table 5.7 illustrates the speed and quality of improvement offered by a ParMETIS load balancing algorithm. It can be considered an extension of Table 5.6 in that the initial entity distribution corresponds to the post-refinement balance in that table. Following those entries is the time required to invoke ParMETIS and physically move data into a more balanced configuration.

Table 5.7 Partition improvements through load balancing

np	32	64	128
before balancing			
minimum node count	1683624	820010	385772
maximum node count	1874249	1023201	581509
node imbalance	**2.7%**	**5.5%**	**10.1%**
minimum tet count	9768743	4731051	2202616
maximum tet count	10896768	5915415	3329890
tet imbalance	**2.7%**	**5.6%**	**10.1%**
time to compute repartition (s)	27.5	19.6	10.4
time to move entities (s)	25.9	22	12.2
after balance			
minimum node count	1689631	819854	385698
maximum node count	1826333	932435	490640
node imbalance	**1.9%**	**3.2%**	**.06%**
minimum tet count	9806471	4730741	2202439
maximum tet count	10602510	5325942	2746515
tet imbalance	**2%**	**3%**	**5.5%**

The actual load balancing calls scale fairly well – approximately 90% scalability for the jump from 64 to 128 processors.

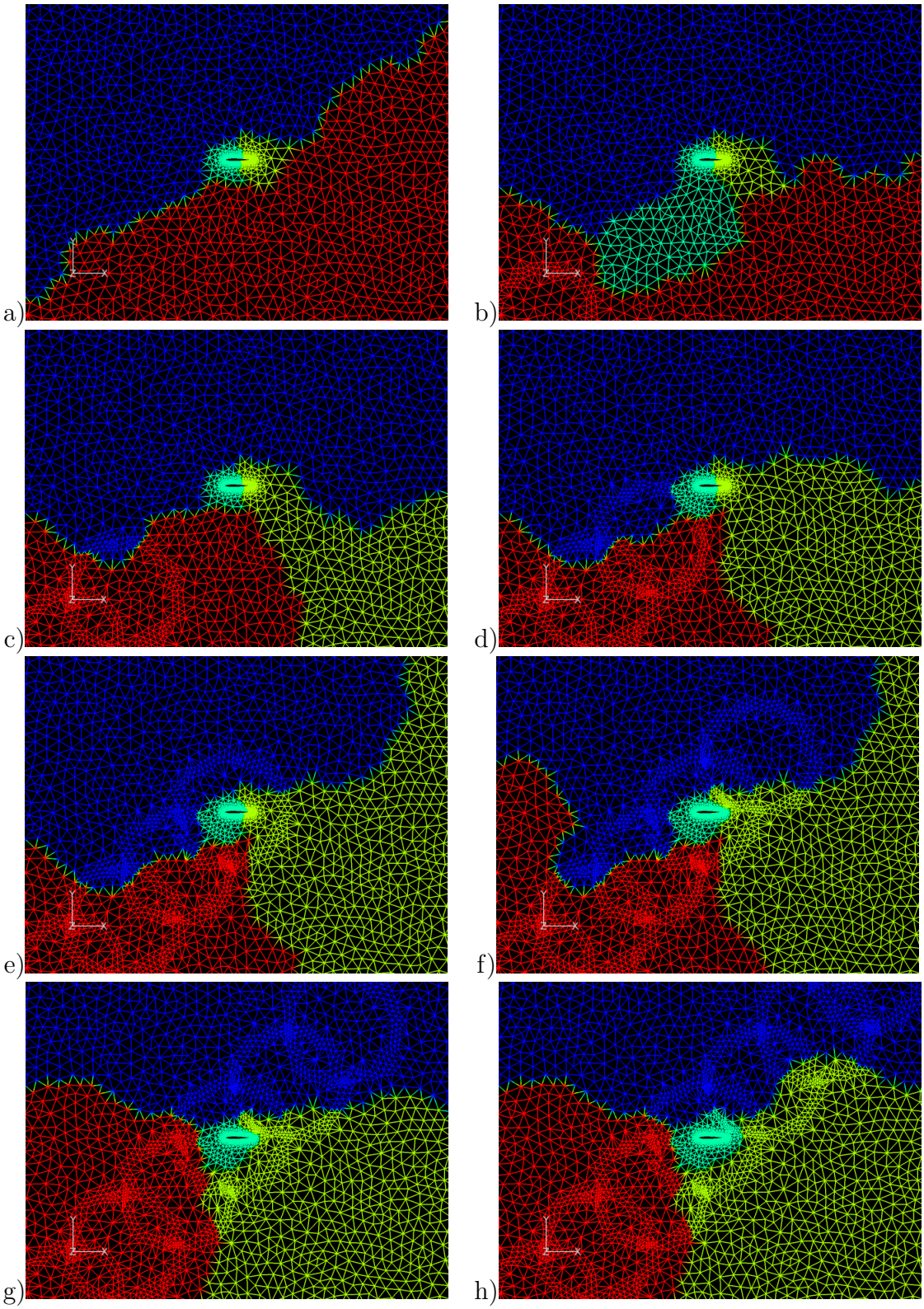

Figure 5.6 Refining and rebalancing the load at each step

5.3.4 Enabling De-refinement

De-refinement, or coarsening, is a much more difficult problem than refinement. The idea is that over the course of an application, it is determined that there are more nodes than necessary in a given region of the mesh, and that excess nodes should be removed. As before, mathematical error estimates could lead to a quantifiable motivation for eliminating nodes.

General mesh coarsening is problematic in that it insinuates that the original mesh generation process was somehow overzealous. In many cases meshes are generated with features designed to serve an application requirement, such as orthogonal packing of mesh entities along surface boundaries, to better resolve viscous effects. General coarsening in these cases could have devastating effects on application results.

Furthermore, general coarsening is difficult. It is a localized mesh generation problem in which nodes are eliminated from a mesh region, and entities are reconstructed from the remaining nodes, using mesh generation techniques (such as Delaunay triangulation).

The approach taken presently is for *de-refinement*: restoring refined mesh entities to an earlier configuration. To facilitate this, we require an elaborate bookkeeping data structure that records all mesh modifications, so that they can be undone.

The de-refinement process proceeds as follows:

1. Identify previous mesh entities to restore.
2. Remove the smaller mesh entities that conflict with the mesh entities being restored.
3. Re-add the original mesh entity to the appropriate indices.
4. Update the bookkeeping data structure.
5. Invoke a restricted mesh refinement process to create transition entities around the newly restored entities as needed.
6. Remove unused nodes (optional, depending on application requirements).

5.3.4.1 De-refinement Bookkeeping

The bookkeeping data structure must be updated every time an entity refinement is made, to include a record of the original entity and its replacements. These replacements may, in turn, be refined and the data structure must be able to reflect this, effectively storing multiple levels of *undo* information.

Additionally, the bookkeeping data structure must be aware of the parallel context of the mesh. Load-balancing often occurs in response to mesh refinement, meaning that nodes and entities are renumbered and moved between processors. The bookkeeping must reflect these numbering changes, including the movement to other processors, so that the processor owning the related entities has the information needed to de-refine.

Lastly, since an overarching *Splatter* goal is to support arbitrary user data, these refined mesh entities may be associated with arbitrary user data that must be maintained along with the bookkeeping records. In other words, entity data also must be restored when a refinement is undone.

Here, we present a solution that accomplishes all of the above by making elaborate use of *Splatter*'s flexible mesh data structures and query syntax. It currently only supports simplicial mesh entities — that is, the de-refinement of tetrahedra and triangles.

The bookkeeping consists of three distinct components:

1. *replacements* – a mapping from original mesh entities to their replacements.
2. *originals* – a list of original mesh entities that may be de-refined.
3. *originals_pending* – a secondary list of original mesh entities that are currently not allowed to be de-refined, because their replacements have been refined. (de-refinement must happen incrementally — one level of undo at a time).

Recall:

1. The lists and mappings comprising the bookkeeping structure must appear on the process that owns the partition containing the related entities.

2. Refinement, de-refinement and load-balancing will occur periodically, meaning that the nodes used in these lists and mappings must be updated.

In short, these bookkeeping data structures must be treated exactly like every other tracked index in use (Section 4.3). The solution: implement these bookkeeping structures as *Splatter* indices.

The framework allows the construction of additional entity types, and does not technically require them to represent polytopes. Here, we introduce a new mesh entity type that represents a replacement entity and its corresponding original.

The *originals* and *pending_originals* indices are `explicit_index` objects of type `topo::TET`. The *replacements* index is an `explicit_index` containing custom entities with 8-node tuples: the first four nodes containing the nodes in the replacement tetrahedron, the last four containing the nodes belonging to the original (see Figure 5.7).

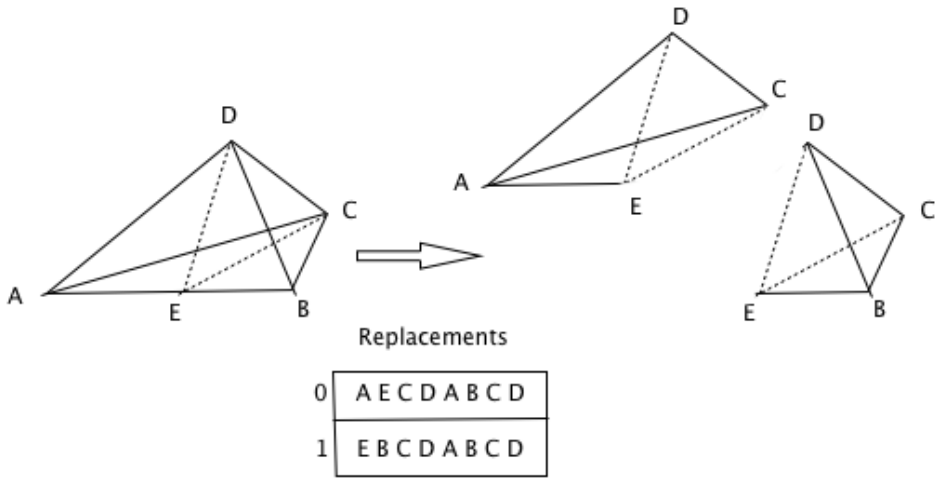

Figure 5.7 De-refinement information stored in a custom `explicit_index`

By tracking these indices with the `part_mgr` (the default behavior for those of type `explicit_index`), they are automatically renumbered and redistributed as appropriate,

whenever any global `renumber/redistribute` occurs, which (as seen in Section 4.3) happens during any operation that impacts the node distribution.

To configure the `part_mgr` to enable de-refinement, a special subroutine called `register_coarsen_handlers` is used. This subroutine registers custom hooks for the refinement process (Section 5.3.1.3), such that whenever entities are refined, the bookkeeping is updated with the proper undo information.

During the de-refinement process, after the entities to be restored are identified, queries are used to find the replacement entities that were derived from these originals. These replacement entities will be removed from the indices and the originals will be put back.

One complication of note: indices that do not represent polytopes, such as the special entities comprising the *replacements* index, are susceptible to *false positives* in the search results, using the standard `fast_index` search structure (Section 4.2.2.2).

In the case of *replacements*, searching for a 4-node tetrahedron would find all 8-node entities that happened to contain all 4 nodes, but those nodes would not be guaranteed to exist entirely in half the tuple corresponding to a specific tetrahedron. For example, 3 of the nodes might be in the part of the tuple representing the original entity, and the 4^{th} might be from the other half.

Because of this possibility, bookkeeping queries that operate on *replacements* must use a secondary test on the results of the search process.

In the current implementation, this secondary test is manifested in a custom query module called `actual_replacements`, which is used in lieu of the standard `faces_in` component. This module outputs only the expected replacement entities (from *replacements*) for an incoming tetrahedron. Listing B contains the heart of this query module. It uses a special `tetmatch` object (an instance of `splatter::picky_matcher` documented in Appendix C) to filter out the false matches from the default search structure.

A final complication of note: user data may be associated with refined mesh entities that must be restored when de-refinement occurs. To account for this, reflective

```
SQ_FILTER(idx, eltno, etype, nnodes, nodes, o)
 {
     // matches gets all replacements returned by
     // the default search structure
     intset matches = replacements.findall(4, nodes);;


     for (unsigned int m=0; m < matches.size(); m++)
     {
         // only output the replacements that meet our secondary
         // screening process
         if (tetmatch(&(replacements[matches[m]][4]), nodes))
         {
             o(&replacements, matches[m],
               topo::TET, 4, replacements[matches[m]]);
         }
     }
 }
```

Listing 5.3 Filter module that verifies the results from the default search structure

properties of the `data_proxy` (see Section 4.2.3.8) are used to clone the data associated with the primary tetrahedra index, attaching compatible data proxies to the *original* and *original_pending* indices. Part of the refinement hooks, as well as the main coarsening algorithm, use the `store_raw` functionality of `data_proxy` to blindly copy associated data between the bookkeeping data structures and user data structures. Thus, arbitrary user data is automatically tracked throughout the refinement/de-refinement process, with no user interaction necessary.

5.3.5 De-refinement Results

With coarsening configured, the refinement test using the artificial sphere refinement function is repeated. Refined entities that fall a distance from the sphere center corresponding to a configurable threshold are de-refined. Figure 5.8 and Figure 5.9 illustrate this process.

Table 5.8 contains the results of this refinement/coarsening demonstration run on the `single_block` mesh introduced in Section 5.2.2. 7 iterations were run, each iteration containing a coarsening stage and a refinement stage.

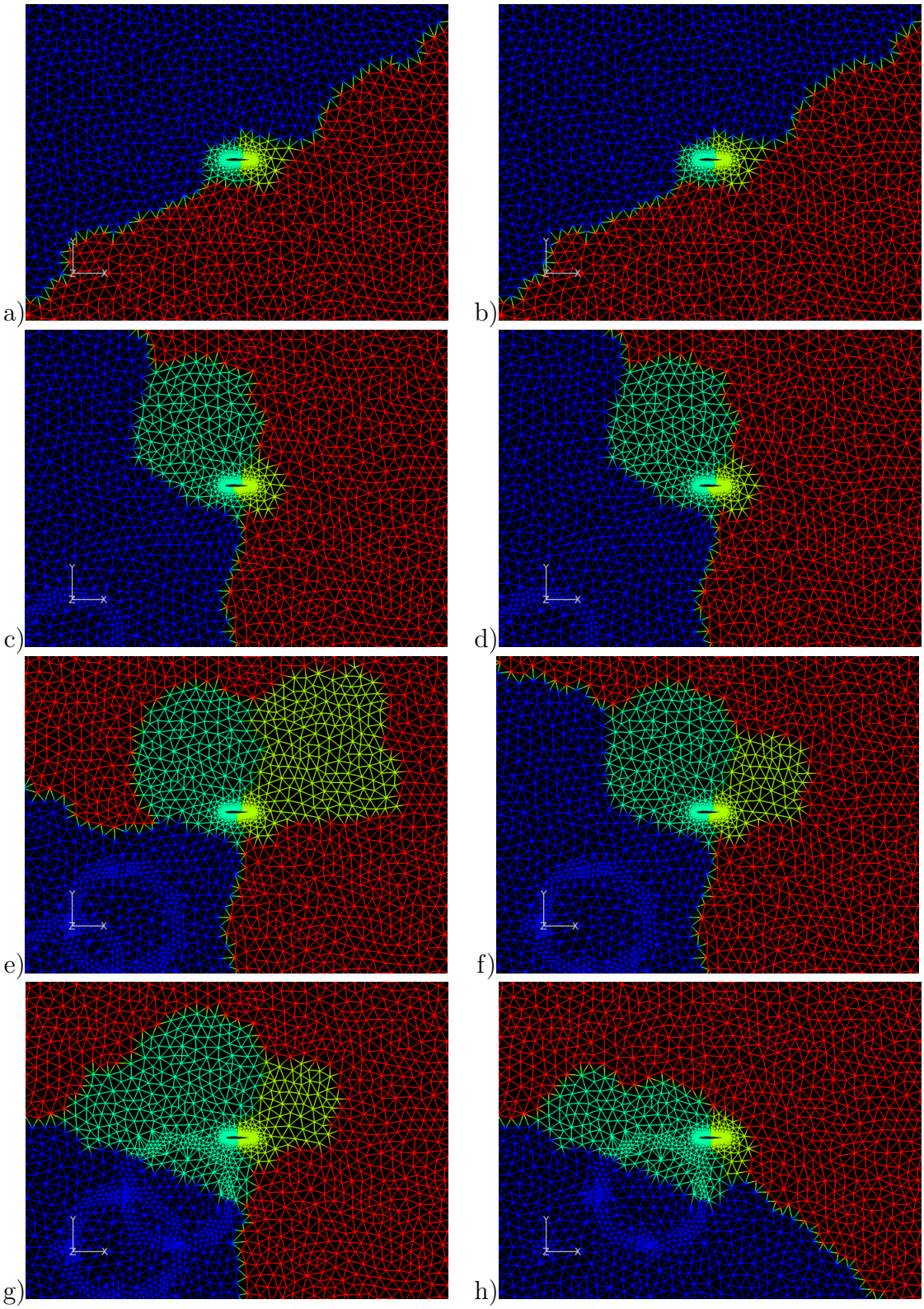

Figure 5.8 Refining, de-refining and rebalancing the load at each step

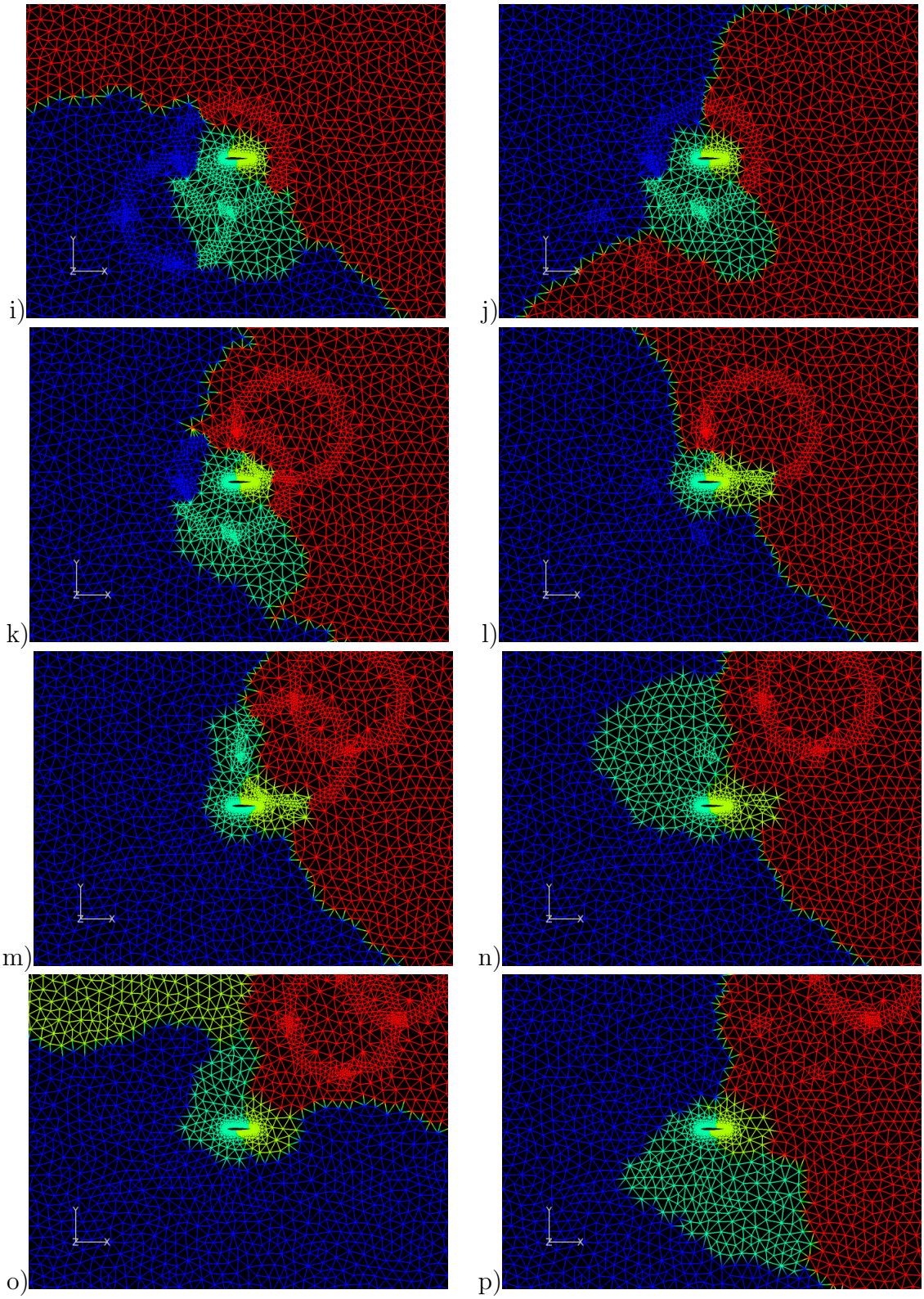

Figure 5.9 Refining, de-refining and rebalancing the load at each step

For each iteration, the following results are reported:

1. entities de-refined: the global number of mesh entities that will be restored. This number does not attempt to count distinct entities; that is, entities shared along partition boundaries will be counted multiple times.
2. de-ref time: the number of seconds to process the de-refinement.
3. entities refined: The initial number of tetrahedron being fully refined. The actual refinement count will be more due to the subsequent calculation of transition entities.
4. refinement time: the number of seconds to process the refinement.
5. node count: the average number of nodes owned per processor.
6. tet count: the average number of volume entities existing on each processor.
7. boundary cont: the average number of triangular boundary entities existing on each processor.
8. phantom node count: the average number of non-local nodes required per processor.
9. memory usage: the maximum size in kilobytes of a single processors virtual memory.
10. iteration time: the wall clock time taken to evaluate the entire process coarsening/refinement

Table 5.8 Refinement and de-refinement of the `single_block` mesh

Processor Count	16	32	64	128
INITIAL				
node count	424885	212442	106221	60825
tet count	2570163	1298160	657397	334037
boundary count	43891	22066	11104	5601

Continued on next page

Table 5.8 Refinement and de-refinement of the `single_block` mesh

Processor Count	16	32	64	128
phantom node count	23438	16943	11498	7715
memory usage (kb)	429456	409060	335372	319796
ITERATION 0				
entities de-refined	0	0	0	0
de-ref time (s)	9.22	3.99	1.84	0.96
entities refined	44749	45841	46877	48169
refinement time (s)	28.30	12.30	6.68	3.48
node count	455271	214952	107476	53738
tet count	2601453	1315408	666479	338203
boundary count	44319	22173	11160	5633
phantom node count	25367	17992	12214	7940
memory usage (kb)	695348	527920	395804	345848
iteration time (s)	53.2	23.8	12.9	7.9
ITERATION 1				
entities de-refined	0	0	0	0
de-ref time (s)	27.39	10.40	5.63	3.46
entities refined	82140	85606	85501	85834
refinement time (s)	34.93	23.10	10.95	5.38
node count	439040	219520	109760	54880
tet count	2656365	1341236	679692	345479
boundary count	44436	22329	11248	5674
phantom node count	26244	17846	12261	8215
memory usage (kb)	821756	600756	421872	365372
iteration time (s)	84.6	46.8	23.0	12.9
ITERATION 2				
entities de-refined	83733	83381	84953	86052
de-ref time (s)	59.61	43.80	21.27	17.50

Continued on next page

Table 5.8 Refinement and de-refinement of the `single_block` mesh

Processor Count	16	32	64	128
entities refined	103518	105863	105648	108613
refinement time (s)	45.72	23.18	11.86	6.60
node count	446676	223338	111669	55835
tet count	2704577	1364338	693405	351507
boundary count	44515	22364	11276	5686
phantom node count	27751	18298	13304	8430
memory usage (kb)	846916	622584	465468	482864
iteration time (s)	124.25	76.59	40.27	29.60
ITERATION 3				
entities de-refined	187082	187772	193030	192870
de-ref time (s)	67.96	46.63	22.03	26.09
entities refined	103792	103582	107598	107693
refinement time (s)	49.49	22.66	13.15	6.15
node count	449575	224788	112394	56197
tet count	2723090	1373950	695970	355070
boundary count	44402	22310	11233	5676
phantom node count	28460	18750	12702	9020
memory usage (kb)	862012	650108	504428	567244
iteration time (s)	147.01	84.34	44.18	37.78
ITERATION 4				
entities de-refined	258287	258644	262169	272252
de-ref time (s)	76.23	36.51	25.64	22.72
entities refined	102874	102681	105170	108317
refinement time (s)	61.59	27.49	11.72	5.98
node count	449491	224747	112375	56186
tet count	2720675	1381067	695882	355052
boundary count	44368	22322	11229	5672

Continued on next page

Table 5.8 Refinement and de-refinement of the `single_block` mesh

Processor Count	16	32	64	128
phantom node count	27776	21472	12725	9027
memory usage (kb)	868604	663836	547936	567244
iteration time (s)	168.97	80.49	48.38	34.47
ITERATION 5				
entities de-refined	261615	269655	266529	274482
de-ref time (s)	125.24	41.51	26.03	20.92
entities refined	102771	106200	106151	111656
refinement time (s)	50.65	32.86	13.20	5.85
node count	44952 0	224762	112385	56196
tet count	2721186	1375045	697439	355542
boundary count	44368	22306	11238	5671
phantom node count	27853	19232	13306	9202
memory usage (kb)	872232	664396	553576	567244
iteration time (s)	204.60	93.77	48.49	32.28
ITERATION 6				
entities de-refined	257427	262433	268309	274919
de-ref time (s)	52.00	73.38	31.32	19.62
entities refined	102411	103727	106727	109618
refinement time (s)	54.25	35.08	13.34	6.39
node count	449772	224902	112452	56230
tet count	2722006	1375929	696357	355523
boundary count	44363	22315	11226	5674
phantom node count	27708	19228	10353	12702
memory usage (kb)	872284	669788	560036	567244
iteration time (s)	137.75	125.31	54.55	31.69

Valuable insight from this table:

1. The refinement algorithm is highly scalable, with an average strong scaling percentage of 101% over the course of these iterations, up to 128 processors.

2. The coarsening algorithm – not quite as impressive as the refinement – with an average strong scaling percentage of 54%. This is a reasonable figure given the complexity of the bookkeeping, as well as the extreme nature of the refinement and coarsening done during the tests.

3. Each iteration roughly results in an equal number of additions and subtractions from the mesh. This is evidenced by the near constant node, tet and boundary counts. Correspondingly, the total memory size remains effectively constant over the runs.

This demonstration and analysis reflects highly usable algorithms. Considering that most real applications will take a far greater amount of time simply to perform application calculations, the time it takes to update and manipulate the mesh will be dwarfed by other concerns in a real application, as seen in the next section.

5.4 Integration with Real Applications

Splatter exists to support real applications. In this section, we discuss the process by which the framework may be integrated with an existing computational fluid dynamics simulation.

An initial 3D CFD flow solver, written in C++, was taken and modified in the following ways:

1. All mesh-IO routines were removed. The original solver required a preliminary offline partitioning process. Framework mesh loading operations, such as those implemented in Appendix A were used instead, allowing the use of standard save/restart capability or on-the-fly load-and-partition of monolithic Star-CD format meshes.

2. All parallel communication related routines were either removed, to be handled automatically by *Splatter*, or replaced by similar framework-aware calls. This part was optional, but done to test framework capabilities.

3. All mesh data structures were replaced, via a global, textual search and replace, with the equivalent `std::vector`'s. This was to enable the direct attachment of user data with existing `splatter::data_proxy` implementations

4. The construction of all ancillary data structures affected by mesh connectivity, such as the compressed row storage indices for the mesh's corresponding linear system, were moved into a single subroutine call that was subsequently registered as a RENUMBER_HOOK (Section 4.3.1) with the `part_mgr`.

Most of these changes were simple textual processes. The more structural modifications were somewhat complicated in that the original solver was written by someone else. The process of modifying it for *Splatter* involved identifying certain data structures that were somewhat obscure. All in all, the process of preparing the solver was straightforward and took a matter of days.

After these changes, the solver ran exactly as before. The only real change to the application process involved the initial mesh loading and partitioning. However, since it was now fully integrated with the framework, all of the queries and modifications mentioned throughout this document were applicable.

Specifically, we added the ability to dynamically refine the mesh in response to flow solution data. This required, in addition to the changes already made, the identification of data refinement rules — that is, the registration of REFINEMENT_HOOKS (Section 5.3.1.3) for passing entity data, such as node coordinates and volume condition tags, to the new nodes and entities introduced during the refinement process.

This was a trivial process, since all of the data relationships were identified during the initial *Splatter* integration process.

5.4.1 Flow Solver Performance

To analyze the performance of the *Splatter*-augmented flow solver, a series of flow simulations were performed on a small mesh with 45372 nodes and 327916 tetrahedra. These tests were executed on 8, 16 and 32 processors.

Figure 5.10 illustrates the results of solution-based refinement. After achieving a first order solution, the flow field was refined by marking cells with high gradients of solution variables, and having *Splatter* fully refine the marked cells. Figure 5.10 demonstrates the results of one of these runs.

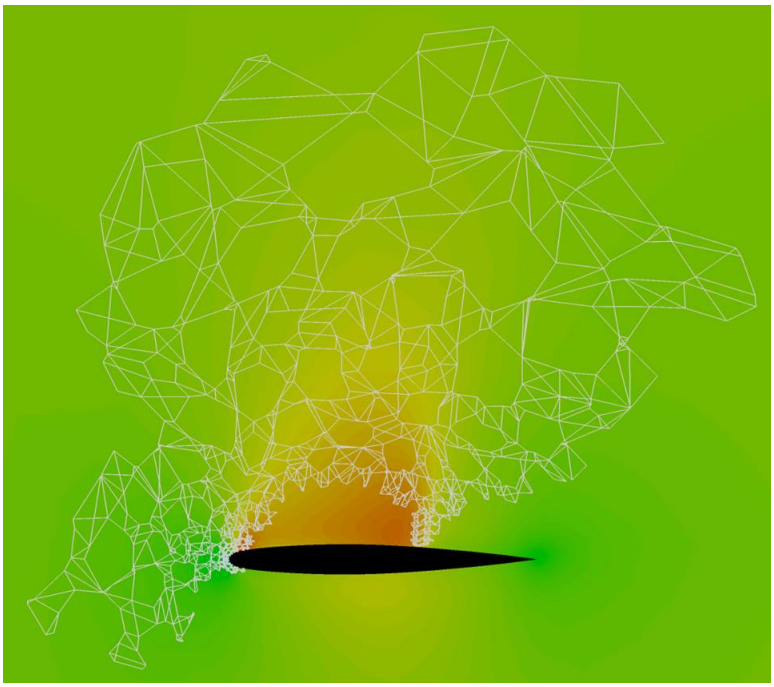

Figure 5.10 Refining the flow field in response to solution variables

For the following tests, a first order solution was achieved prior to the timed tests. The following results indicate the relative performance of the solver executing several iterations of a 2^{nd} order solution process.

Table 5.9 contains average iteration times in wall clock seconds for this 2^{nd} order process, run on a variety of processor counts.

Table 5.9 Solver times on various processor counts

np	8	16	32	strong scaling
base time	2.73	1.62	1.07	0.63785047
time with refinement	12.18	6.56	3.71	0.82075472
time with refinement and load balancing	5.8	3.06	1.81	0.80110497

The specific mesh refinement case executed introduced a significant number of new nodes due to the particular refinement function invoked. This accounts for the significant performance drop between the base fun time and the run with full refinement and load balancing. Furthermore, it is clear that refinement without load balancing is a bad idea.

Specifically, Table 5.10 contains the before and after average local mesh sizes from the load-balancing process, to gauge the amount of extra work introduced by the refinement process. It also contains the average number of phantom nodes before and after the load-balancing process.

Lastly, it contains the time taken to execute the refinement and load balancing algorithms.

Considering the relatively high level of refinement in these cases, *Splatter*'s contribution to the total runtime is low. The load-balancing functionality, in particular, provides a major service using a minimal amount of time.

These results provide some context for the performance analyses in the previous sections. Compared to typical application work, the framework's manipulation of the mesh takes a small amount of time.

Table 5.10 Mesh entities created by refinement, and time required

np	8	16	32
initial node count	5672	2831	1418
initial phantom node count	762	572	452
initial tet count	31703	16329	8560
refinement time (s)	.89	.62	.47
phantom node count (pre load-balance)	1137	883	702
load balancing time (s)	.70	.63	.54
final node count	11927	5936	2982
final phantom node count	1213	1049	756
final tet count	68482	35186	18275

CHAPTER 6
CONCLUSION

Splatter is a new mesh management framework characterized by extreme flexibility and easy integration with application codes.

The unique embedded query syntax provides a high level of expressibility, leading to increased correctness and ease of development. The query system is highly optimized C++ code, so this flexibility does not come with a penalty on performance.

The ability to programmatically define mesh entity types, along with the existence of these queries, presents a general unstructured mesh as a database. It is hoped that the inherent flexibility in this configuration will enable experimental applications beyond the traditional scientific uses of unstructured meshes.

The framework's data model, based on the use of proxies, untethers framework logic from the underlying data structures. The end result is a powerful model for data manipulation that assures data integrity despite complex mesh manipulations and parallel redistributions.

It is this same notion of proxies that allows for the straightforward integration with application codes. By attaching to existing user data structures, the framework can manipulate user data without requiring ubiquitous and cryptic API calls.

The design and implementation of this framework has been discussed in detail, and specific applications of the framework have been demonstrated. In particular, the framework has been used for refinement and de-refinement of unstructured meshes, along with parallel load balancing. Performance analysis has been very encouraging.

Finally, the integration of the framework with a working 3D computational flow solver has been discussed, proving that the framework is suitable for real scientific applications.

6.1 Future Work

The current *Splatter* implementation is hopefully the starting point for many interesting future projects.

Most generally, since the framework is based on a standard C++ class model, additional advanced capabilities can be added to the framework through the extension of classes. Particularly, new index types can be designed that use different search structures, optimized for different algorithms, and new data proxies can be implemented to further the integration capabilities with existing codes.

Of particular interest is a data proxy capable of operating on Fortran data. This, it is assumed, will be one of the first such extensions.

Additionally, the query syntax can be improved without breaking existing compatibility. The most pressing improvement to the query system is the existence of hybrid parallel (multithreaded) queries, as multi-core hardware is now commonplace.

Also, a tighter integration with mesh data would allow more functionality to be implemented directly in the query syntax, instead of in surrounding code or custom query modules. A static query analyzer that helps in writing high performance queries — or at least interpreting cryptic compiler errors — would be very welcome.

Other future ideas relate to the tightening integration of *Splatter* with existing software libraries. PETSc [35] is a remarkable tool for solving nonlinear equations and linear systems in parallel. Integrating PETSc's solvers into *Splatter* queries would allow for the rapid development of new simulation codes.

Similarly, several movements are afoot to define the interfaces between interchangeable scientific software components. Two of note are ITAPS [17] and the Common Component Architecture[36]. It would be a boon to have *Splatter* integration with either of these projects.

Splatter provides an efficient implementation of a very flexible design. Hopefully it will be used for some interesting applications. The specifics of those applications will define the future of this software.

REFERENCES

[1] Oliker, L., Biswas, R., and Gabow, H. N., "Parallel tetrahedral mesh adaptation with dynamic load balancing," *Parallel Computing*, Vol. 26, No. 12, 2000, pp. 1583–1608.

[2] Garimella, R., "A Practical Guide to Developing and Using Mesh and Geometry Frameworks for Advanced Meshing and Computational Software," *International Meshing Roundtable*, Oct. 2011, pp. 1–62.

[3] De Floriani, L. and Hui, A., "Shape representation based on simplicial and cell complexes," *Eurographics 2007 - State of the Art Reports*, 2007, pp. 63–87.

[4] Kremer, M., Bommes, D., and Kobbelt, L., "OpenVolumeMesh–A Versatile Index-Based Data Structure for 3D Polytopal Complexes," *Proceedings of the 21st International Meshing Roundtable*, 2012, pp. 531–548.

[5] Garimella, R. V. and Shephard, M. S., "Boundary layer mesh generation for viscous flow simulations," *International Journal for Numerical Methods in Engineering*, Vol. 49, No. 1, 2000, pp. 193–218.

[6] Kapadia, S., *Computational Design and Sensitivity Analysis of Solid Oxide Fuel Cells*, Ph.D. thesis, University of Tennessee at Chattanooga, July 2008.

[7] McKenney, P. E., *Is Parallel Programming Hard, And, If So, What Can You Do About It?*, kernel.org, Aug. 2012.

[8] Seol, E. S., *FMDB: Flexible Distributed Mesh Database for Parallel Automated Adaptive Analysis*, Ph.D. thesis, Rensselaer Polytechnic Institute, Nov. 2005.

[9] Beall, M. and Shephard, M. S., "A general topology-based mesh data structure," *International Journal for Numerical Methods in Engineering*, Vol. 40, No. 9, 1997, pp. 1573–1596.

[10] Zhou, M., Xie, T., Seol, E. S., Shephard, M. S., Sahni, O., and Jansen, K. E., "Tools to support mesh adaptation on massively parallel computers," *Engineering with Computers*, Vol. 28, No. 3, April 2011, pp. 287–301.

[11] Shephard, M. S., Seol, E. S., Smith, C., Mubarak, M., Ovcharenko, A., and Sahni, O., "Methods and Tools For Parallel Anisotropic Mesh Adaptation And Analysis," *Proceedings of the VI International Conference on Adaptive Modeling and Simulation*, May 2013.

[12] Zhou, M., Sahni, O., Shephard, M. S., Carothers, C., and Jansen, K. E., "Adjacency-based data reordering algorithm for acceleration of finite element computations," *Scientific Programming*, Vol. 18, No. 2, 2010, pp. 107–123.

[13] Xie, T., *Mesh Data Management Components for Petascale Adaptive Unstructured Mesh Based Simulations*, Ph.D. thesis, Rensselaer Polytechnic Institute, 2012.

[14] Devine, K. D., Diachin, L., Kraftcheck, J. A., Jansen, K. E., Leung, V., Luo, X., Miller, M., Ollivier-Gooch, C., Ovcharenko, A., Sahni, O., Shephard, M. S., Tautges, T. J., Xie, T., and Zhou, M., "Interoperable mesh components for large-scale, distributed-memory simulations," *Journal of Physics: Conference Series*, Vol. 180, Aug. 2009, pp. 012011.

[15] Rensselaer Polytechnic Institute, *The FMDB User's Guide*, Aug. 2012.

[16] Ledoux, F., Weill, J.-C., and Bertrand, Y., "GMDS: A Generic Mesh Data Structure," *17th International Meshing Roundtable, United States*, 2008.

[17] "ITAPS Interfaces," `http://www.itaps-scidac.org/software/download_interfaces.html`, 2011, [Online; accessed 8-July-2011].

[18] Tautges, T. J., Ernst, C., Stimpson, C., Meyers, R. J., and Merkley, K., "MOAB: a mesh-oriented database." Tech. rep., Sandia National Laboratories, April 2004.

[19] Garimella, R., "MSTK-a flexible infrastructure library for developing mesh based applications," *Proceedings of 13th International Meshing Roundtable*, 2004, pp. 203–212.

[20] "STK-Home," `http://trilinos.sandia.gov/packages/stk/`, 2013, [Online; accessed 9-May-2013].

[21] Kirk, B. S., Peterson, J. W., Stogner, R. H., and Carey, G. F., "`libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations," *Engineering with Computers*, Vol. 22, No. 3–4, 2006, pp. 237–254, `http://dx.doi.org/10.1007/s00366-006-0049-3`.

[22] "Mesh Generation Software for CFD - Pointwise," `http://www.pointwise.com`, 2013, [Online; accessed 30-Sep-2013].

[23] "CFD General Notation System," `http://cgns.org`, 2013, [Online; accessed 25-Aug-2013].

[24] The MPI Forum, C., "MPI: a message passing interface," *Proceedings of the Conference on High Performance Networking and Computing*, 1993, pp. 878–883.

[25] Segal, J., "SECSE: Some Challenges Facing Software Engineers Developing Software for Scientists," *Software Development Processes for Computational Science and Engineering*, March 2009, pp. 9–14.

[26] Celes, W., Paulino, G. H., and Espinha, R., "A compact adjacency-based topological data structure for finite element mesh representation," *International Journal for Numerical Methods in Engineering*, Vol. 64, No. 11, 2005, pp. 1529–1556.

[27] McGuire, M., "The Half-Edge Data Structure," `http://www.flipcode.com/archives/The_Half-Edge_Data_Structure.shtml`, 2013, [Online; accessed 30-Sep-2013].

[28] Karypis, G. and Kumar, V., "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, Vol. 20, No. 1, Dec. 1998.

[29] Schloegel, K., Karypis, G., and Kumar, V., "Graph partitioning for high-performance scientific simulations," *Sourcebook of parallel computing*, Jan. 2003.

[30] Catalyurek, U. V. and Aykanat, C., "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 7, July 1999.

[31] Teresco, J., Devine, K. D., and Flaherty, J., "Partitioning and dynamic load balancing for the numerical solution of partial differential equations," *Numerical solution of partial differential equations on parallel computers*, 2006, pp. 55–88.

[32] Devine, K. D., Boman, E. G., Riesen, L., Catalyurek, U. V., and Chevalier, C., "Getting started with zoltan: A short tutorial," *Proc. Dagstuhl Seminar Combinatorial Scientific Computing, Also Sandia National Labs Tech Report SAND2009-0578C*, 2009.

[33] "ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering," `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`, 2013, [Online; accessed 6-Sep-2013].

[34] Frey, P.-J. and Alauzet, F., "Anisotropic mesh adaptation for CFD computations," *Computer methods in applied mechanics and engineering*, Vol. 194, No. 48, 2005, pp. 5068–5082.

[35] Balay, S., Brown, J., , Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H., "PETSc Users Manual," Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

[36] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B., "Toward a common component architecture for high-performance scientific computing," *Proceedings of the 8th International Symposium on High Performance Distributed Computing*, 1999, pp. 115–124.

APPENDIX A
DEMONSTRATION APPLICATION

```
1  // splatter refine/coarsening demo
2
3  #include <splatter.h>
4  #include <splatt_query.h>
5  #include <splatt_coarsen.h>
6  #include <splatt_geom.h>
7  #include <unistd.h>             // for getopt
8
9  using namespace splatter;
10
11
12 // This custom query module is used to mark tets for refinement
13 class on_radius : public query::query_op
14 {
15 public:
16     on_radius(const array<double,3> center,
17               const double radius,
18               const std::vector<starcd::node_coords>& coords) :
19         center(center), radius2(radius*radius), coords(coords)
20     {}
21
22
23     // return true if tet should be refined
24     // SQ_COND makes this query module usable in a 'where' clause
25     SQ_COND(idx, eltno, etype, nnodes, nodes)
26     {
27         // does the radius arc through this face?
28         bool has_closer_node  = false;
29         bool has_farther_node = false;
30
31         for (int n=0; n<nnodes && !(has_closer_node && has_farther_node);
               n++)
32         {
33             // get distance from center, squared
34             int gn = (*odbp).g2l_p(nodes[n]);
35             bool farther = (geom::dist2(coords[gn], center)-radius2)>0;
36
37             if (farther)
38             {
39                 has_farther_node = true;
40             }
41             else
42             {
43                 has_closer_node = true;
44             }
45         }
46
47         if (has_farther_node && has_closer_node)
48         {
49             return true;
```

```
50          }
51          else
52          {
53              return false;
54          }
55      }
56
57
58  private:
59      ownerdb* odb;
60      array<double, 3> center;
61      double radius2;
62      const std::vector<starcd::node_coords>& coords;
63  };
64
65
66  // This custom query module is used to mark tets for de-refinement
67  class outside_radius : public query::query_op
68  {
69  public:
70      outside_radius(const array<double,3> center,
71                     const double radius_in,
72                      const double radius_out,
73                     const std::vector<starcd::node_coords>& coords) :
74          center(center), radiusin2(radius_in*radius_in),
75          radiusout2(radius_out*radius_out),coords(coords)
76      {}
77
78
79      SQ_COND(idx, eltno, etype, nnodes, nodes)
80      {
81          // are all nodes wihtin the radius?
82          bool all_closer=true;
83          bool all_farther=true;
84
85          for (int n=0; n<4; n++)
86          {
87              // get distance from center, squared
88              int gn = (*odbp).g2l_p(nodes[n]);
89
90              if (gn == -1)       // this is so that the refinement process
                    will
91                                  // get rid of this
92              {
93                  return true;
94              }
95
96              double dist=geom::dist2(coords[gn], center);
97
98              bool closer = dist<radiusin2;
99              bool farther =dist>radiusout2;
```

```
100
101             all_closer &= closer;
102             all_farther &= farther;
103         }
104
105
106         return all_closer | all_farther;
107     }
108
109
110
111 private:
112     ownerdb* odb;
113     array<double, 3> center;
114     double radiusin2, radiusout2;
115     const std::vector<starcd::node_coords>& coords;
116 };
117
118 // include some uninteresting query modules for calculating mesh volumes
119 #include "volparts.cpp"
120
121 // Hook argument for tet refinement
122 struct tet_ref_arg
123 {
124     std::vector<starcd::tet_cond_data>* vcondp;
125 };
126
127
128 // Hook argument for triangle refinement
129 struct tri_ref_arg
130 {
131     std::vector<starcd::bnd_cond_data>* bcondp;
132 };
133
134
135 // Hook argument for node refinement
136 struct node_ref_arg
137 {
138     std::vector<starcd::node_coords>* coordsp;
139 };
140
141
142 // forward declaration of refinement hooks
143 void new_tet_cond(part_mgr* mgr, void* edgesp, void* udata);
144 void new_node_coords(part_mgr* mgr, void* edgesp, void* udata);
145
146
147 // TODO: make this not global
148 std::vector<int> tet_is_refined;
149
150
```

```
151 // forward declaration of demo-specific utility for calculating "big
       circle" radius
152 void get_bounds(const std::vector<starcd::node_coords>& nc, const
       parallel_ctx& ctx,
153                 double* lowxp, double* hixp,
154                 double* lowyp, double* hiyp,
155                 double* lowzp, double* hizp);
156
157
158
159 // parameters, set by getopt, configuring this demo invocation
160
161 struct runcfg
162 {
163     bool do_load;
164     bool do_restart;
165     bool do_balance;
166     bool init_decomp;
167     std::string project;
168     bool do_save;
169     bool do_init_save;
170     bool do_final_save;
171     std::string save_name;
172     bool do_coarsen;
173     int num_iterations;
174     int num_loaders;
175     int coarsen_interval;
176     int refine_interval;
177     int subdivs;                // inverse speed of refinement front
178     double circle_rad;          // circle radius
179     double coarsen_thresh;
180     int fast_forward;
181 };
182
183
184 static const char*  usage_str =
185     "-l  <int>  -- number of loaders of starcd project\n"
186     "-p <projectname> -- starcd project to load\n"
187     "-r <restartname> -- restart file to load (incompatible with -p)\n"
188     "-o <restartname> -- output restart filename\n"
189     "-i <int> -- number of iterations\n"
190     "-S -- save decomp'd version after loading\n"
191     "-F -- save final version at end of iterations\n"
192     "\n"
193     "-c -- enable coarsening\n"
194     "-b -- disable load balancing\n"
195     "-B -- disable initial partitioning\n"
196     "\n"
197     "-C <int> -- coarsening interval (default to 1)\n"
198     "-R <int> -- refinement interval (default to 1)\n"
199     "\n"
```

```
200     "-v <int> -- subdivision of xy area to calculate circle xy delta\n"
201     "-d <double> -- circle radius\n"
202     "-t <double> -- coarsening threshold\n"
203     "-f <int> -- fast forward refinement iterations\n"
204     ;
205
206
207 // forward declaration of routine that handles options
208 status configure_run(runcfg& cfg, int argc, char** argv, int numload);
209
210
211
212
213 // Without further ado,  the main routine...
214
215
216 int main(int argc, char** argv)
217 {
218     MPI_Init(&argc, &argv);
219
220     // 'big_circle' parameters
221     double zero[] = { -5.0, -5,0, 0.0 };
222     double p110[] = { 1, 1, 0 };
223
224     // set up circles
225     array<double,3> circle_center = zero;
226     array<double,3> circle_delta = p110;
227
228     double circle_refine_radius= 5;
229     double circle_coarsen_thresh = .1;
230
231     // filename used for restart files
232     std::string outfile;
233
234
235     // the starcd loader will populate these
236     std::vector<starcd::node_coords>    coords; // node coordinates
237     std::vector<int>                    tets;   // tet nodes
238     std::vector<starcd::tet_cond_data>  vcond;  // tet volume conditions
239     std::vector<int>                    bnds;   // boundary (triangle)
            nodes
240     std::vector<starcd::bnd_cond_data>  bcond;  // boundary conditions
241
242
243     LLOG(0, "in the beginning....");
244
245
246     // this guy is in charge of everything -- load it with the
247     // standard mesh entities, default MPI configuration,
248     // and initialize immediately
249     part_mgr mgr(topo::num_std_entities, topo::std_entities);
```

```
250
251
252     // configure this run using getopt
253     runcfg cfg;
254     if (configure_run(cfg, argc, argv, mgr.pctx().np()) != SPLATT_OK)
255     {
256         if (mgr.pctx().rank() == 0)
257         {
258             std::cout << usage_str <<std::endl;
259         }
260
261         ERROR("error configuring run");
262     }
263
264
265     // runtime configuration complete
266     // -------------
267
268
269     // Timers to analyze performance of routines
270     timer total_timer;
271     timer part_timer;
272
273
274     // reroute std(err|out) to files
275     if (mgr.pctx().np() > 1)
276     {
277         mgr.pctx().rebind_stdio();
278     }
279
280
281
282     int namelen=0;
283     char proc_name[MPI_MAX_PROCESSOR_NAME];
284     MPI_Get_processor_name(proc_name, &namelen);
285
286     LOG("rank " << mgr.pctx().rank() << " of " << mgr.pctx().np() << " on
            " << proc_name);
287     LOG("project: " << cfg.project);
288     LOG("nloaders: " << cfg.num_loaders);
289     LOG("num iterations: " << cfg.num_iterations);
290     LOG("coarsening: " << (cfg.do_coarsen ? "true" : "false"));
291     LOG("load balancing: " << (cfg.do_balance ? "true" : "false"));
292     LOG("initial decomp: " << cfg.init_decomp);
293
294
295     // TBL is a macro that logs information in a format that can easily be
296     // converted to a CSV file using an offline script provided as part of
            the
297     // framework
298     TBL(-1, "rank", mgr.pctx().rank());
```

```
299
300
301     // Configure indices
302     // ------------------
303     // this sets up the initial node ownership
304     splatter::index* idx = mgr.get_node_index();
305
306     // register node data
307     idx->attach_data( "coords", splatter::proxy (coords));
308
309     // set up tet index from vector loaded above; this index introduces
310     // phantom dependencies, and should have a fast node search.
311     idx = mgr.add_explicit_index("tets", topo::TET, tets,
312                                  SPLATT_PHANTOM_DEPS | SPLATT_FAST_INDEX);
313     // attach the tet data vector to this index
314     idx->attach_data("vcond", splatter::proxy(vcond));
315
316
317     // set up boundary index from vector loaded above
318     // no phantom deps needed, since these are handled by the tets, above
319     idx = mgr.add_explicit_index("bnds", topo::TRI, bnds,
        SPLATT_FAST_INDEX);
320
321     // attach bnd data vector to this index
322     idx->attach_data("bcond", splatter::proxy(bcond));
323
324
325
326     // external data structure maintaining de-refinement bookkeping data
327     coarsen_kernel* coarsenk=NULL;
328
329
330     if (cfg.do_coarsen)
331     {
332         // register appropriate refinement hooks with the framework, to
            handle
333         // the updating of de-refinement bookkeeping
334         coarsenk = register_coarsening_handlers(&mgr,
335                                                 mgr.get_index("tets"),
336                                                 mgr.get_index("bnds"));
337
338     }
339
340
341
342
343
344     // load the mesh on cfg.nloaders processors
345
346
347     bool involved_in_init=false;
```

```
348
349     MPI_Group fullgroup;
350     MPI_Group loadgroup;
351     MPI_Comm loadercomm;
352     MPI_Comm_group(mgr.pctx().comm(), &fullgroup);
353
354     // temporary parallel context for file load
355     parallel_ctx loader_ctx;
356
357
358     if (cfg.do_restart)
359     {
360         // load mesh and data from restart files with name cfg.project
361
362         part_timer.start();
363         int onp = mgr.load(cfg.project);
364
365         LOG("TIME: initial restart load took " << part_timer.time());
366         TBL(-1, "*restart_load", part_timer.time());
367
368         // log diagnostics about virtual memory usage
369         LOG_MEM(-1, "post_load_restart");
370
371         // error code
372         if (onp == 0)
373         {
374             ERROR("invalid restart project: " << cfg.project);
375         }
376
377         if (onp == mgr.pctx().np())
378         {
379             // the restart files were loaded by the same number of
                    processors
380             // that created them
381
382             LOG("no need to rebalance after restart");
383             cfg.init_decomp=false;
384         }
385         else
386         {
387             // the restart files were loaded on more processors than
                    orginally
388             // created them, so they must be repartitioned
389
390             LOG("need to rebalance: " << onp << " of " << mgr.pctx().np()
                    << " had data");
391             if (mgr.pctx().rank() < onp)
392             {
393                 involved_in_init = true;
394             }
395
```

```
396             // create an MPI_COMM for the ranks involved in initial IO/
                   parmetis
397             std::vector<int> loaderranks(onp);
398             for (unsigned int i=0; i < loaderranks.size(); i++)
399             {
400                 loaderranks[i]=i;
401             }
402
403             // create MPI communicator containing processors with valid
                   mesh data
404             MPI_Group_incl(fullgroup, onp, &loaderranks[0], &loadgroup);
405             MPI_Comm_create(mgr.pctx().comm(), loadgroup, &loadercomm);
406         }
407
408     }
409     else
410     {
411         // ------------------
412         // LOAD Monolithic mesh files in parallel
413
414         // create an MPI_COMM for the ranks involved in initial IO/
               parmetis
415         std::vector<int> loaderranks(cfg.num_loaders);
416         for (unsigned int i=0; i < loaderranks.size(); i++)
417         {
418             loaderranks[i]=i;
419         }
420
421         if (mgr.pctx().rank() < cfg.num_loaders)
422         {
423             involved_in_init = true;
424         }
425
426         MPI_Group_incl(fullgroup, cfg.num_loaders, &loaderranks[0], &
               loadgroup);
427         MPI_Comm_create(mgr.pctx().comm(), loadgroup, &loadercomm);
428
429         part_timer.start();
430         if (involved_in_init)
431         {
432             loader_ctx.init(loadercomm);
433             assert(loader_ctx.rank() == mgr.pctx().rank());
434
435             part_timer.start();
436             LLOG(0, "loading data files");
437
438             // call the starcd routines
439             if (starcd::load_all(cfg.project, loader_ctx, coords, tets,
                   vcond, bnds, bcond)
440                 == SPLATT_FAIL)
441             {
```

```
442                 ERROR("failed to load "<<cfg.project<<" files");
443                 MPI_Abort(mgr.pctx().comm(), -1);
444             }
445             LLOG(0, "file loading took " << part_timer.time());
446             TBL(-1, "*project_load", part_timer.time());
447         }
448         else
449         {
450             TBL(-1, "*project_load", -1);
451         }
452 
453 
454         // non-loaders wait here....
455         MPI_Barrier(mgr.pctx().comm());
456         LOG("TIME: initial load took " << part_timer.time());
457 
458         LLOG(0, "files loaded on " << cfg.num_loaders << " of " << mgr.
               pctx().np() << " nodes");
459         LOG_MEM(-1, "post_load_proj");
460         LLOG(0, "number of coords: " << coords.size());
461         LLOG(0, "number of tets: " << (tets.size()/4));
462         LLOG(0, "number of bnds: " << (bnds.size()/3));
463 
464         TBL(-1, "coords_pref", coords.size());
465         TBL(-1, "tets_pref", (tets.size()/4));
466         TBL(-1, "bnds_pref", (bnds.size()/3));
467 
468 
469         // mesh data is loaded on first nloader ranks
470 
471         // build node distribution out of local vector sizes
472         mgr.config_node_index(coords.size());
473 
474 
475         part_timer.start();
476 
477         // route all entities to processors that own the corresponding
               nodes
478         mgr.finalize_load();
479 
480         LOG("TIME: finalize_load() took " << part_timer.time());
481         TBL(-1, "*finalize_load", part_timer.time());
482     }
483 
484 
485     LLOG(0, "log/finalize complete");
486     LOG_MEM(-1, "predecomp");
487     LLOG(0, "number of coords: " << coords.size());
488     LLOG(0, "number of tets: " << (tets.size()/4));
489     LLOG(0, "number of bnds: " << (bnds.size()/3));
490 
```

```
491     TBL(-1, "coords_load", coords.size());
492     TBL(-1, "tets_load", (tets.size()/4));
493     TBL(-1, "bnds_load", (bnds.size()/3));
494
495
496     // The mesh is loaded.  Do initial partitioning if needed.
497
498     if (cfg.init_decomp)
499     {
500         int edgecut;
501         std::vector<int> new_numbering;
502         std::vector<int> new_node_dist;
503
504         LLOG(0, "initiating [or waiting for] initial decomp");
505         part_timer.start();
506         if (involved_in_init)
507         {
508             // call ParMETIS routines to generate new node numbering and
                    distribution
509
510             decomp(loadercomm, mgr.get_index("tets"), mgr.pctx().np(), &
                    edgecut,
511                     mgr.get_ownerdb(),
512                     new_numbering, new_node_dist, false, -1);
513
514         }
515         else
516         {
517             LOG("TIME: parmetis call took -1");
518             TBL(-1, "*parmetis", -1);
519         }
520
521         LOG("TIME: decomp call took " << part_timer.time());
522         TBL(-1, "*decomp", part_timer.time());
523
524         MPI_Barrier(mgr.pctx().comm());
525
526         part_timer.start();
527
528         {
529             // communicate new node distribution with processors
                    uninvolved in
530             // decomp
531             mgr.pctx().broadcast(new_node_dist);
532
533             // invoke the global mesh renumber/redistribute routines to
                    update the entire mesh
534             LLOG(0, "initial renumbering of post-decomp'd mesh");
535             mgr.renumber(new_numbering, new_node_dist);
536         }
```

```
537         LOG("TIME: initial renumber (for decomp) took " << part_timer.time
                ());
538         TBL(-1, "*renumber_init", part_timer.time());
539     }
540
541
542     MPI_Barrier(mgr.pctx().comm());
543     part_timer.start();
544     LLOG(0, "augmenting nodes at end of iteration");
545
546     // build phantom node maps and update phantom node data
547     mgr.augment_nodes();
548     LOG("TIME: augment_nodes took " << part_timer.time());
549     TBL(-1, "*augment_init", part_timer.time());
550
551     LOG_MEM(-1, "post_augment1");
552
553     LLOG(0, "ready to run");
554     LLOG(0, "initial node dist: " << mgr.get_ownerdb());
555     LLOG(0, "initial tet dist: " << mgr.get_index("tets")->size());
556     LLOG(0, "initial bnd dist: " << mgr.get_index("bnds")->size());
557
558     TBL(-1,"nlocal_init", mgr.get_ownerdb().nlocal());
559     TBL(-1, "coords_init", coords.size());
560     TBL(-1, "tets_init", (tets.size()/4));
561     TBL(-1, "bnds_init", (bnds.size()/3));
562
563
564
565     LLOG(0, "initial decomp/data distribution complete");
566
567
568     mgr.pctx().barrier();
569
570
571     // save post-decomp restart files if desired
572
573
574     if (cfg.do_init_save)
575     {
576         outfile = cfg.save_name + "init";
577         mgr.save_start(outfile);
578         mgr.save_node_data("coords");
579         // mgr.save_data("tets", "refstat");
580         mgr.save_end();
581     }
582
583
584     // stop here, unless we are actually running the refinement demo
585     if (cfg.num_iterations <= 0)
586     {
```

```
587         LOG("the end (no iterations)");
588         MPI_Finalize();
589         exit(0);
590     }
591
592
593     // register refinement hooks
594
595 // uncomment these to update tet and boundary data during refinement
596 //    mgr.register_hook(SPLATT_REFINE_HOOK, topo::TRI, hookify(
        new_bnd_cond, &bcond));
597 //    mgr.register_hook(SPLATT_REFINE_HOOK, topo::TET, hookify(
        new_tet_cond, &vcond));
598     mgr.register_hook(SPLATT_REFINE_HOOK, topo::NODE, hookify(
            new_node_coords, &coords));
599
600     explicit_index* tet_idx = mgr.get_index("tets");
601     explicit_index* bnd_idx = mgr.get_index("bnds");
602
603     // refinement flag to avoid too much refinement in this demo
604     tet_idx->attach_data("tet_is_refined", proxy(tet_is_refined));
605
606     if (cfg.do_coarsen)
607     {
608         // make sure the "tet_is_refined" data is restored when de-
                refining
609         coarsen_track_tet(coarsenk, "tet_is_refined");
610     }
611
612
613
614     part_timer.start();
615     LLOG(1, "building fast indices");
616
617     // pre-build search structures.
618
619     mgr.build_fast_indices();
620     LOG("TIME: build_fast_indices took "<< part_timer.time());
621     TBL(-1, "*build_fast_idx", part_timer.time());
622
623
624     // set up circle based on actual dimensions of mesh and runtime
            configuration
625     double lowx, lowy, lowz, hix, hiy, hiz;
626     get_bounds(coords, mgr.pctx(), &lowx, &hix, &lowy, &hiy, &lowz, &hiz);
627     double delx = hix-lowx, dely=hiy-lowy, delz=hiz-lowz;
628
629     circle_center[0] =  lowx;
630     circle_center[1] =  lowy;
631     circle_center[2] =  delz/2.0;
632     double smallest = delx;
```

```
633     if (dely < smallest) smallest = dely;
634     if (delz < smallest) smallest = delz;
635
636     if (cfg.circle_rad < 0)
637     {
638         circle_refine_radius = delx/2.1;
639     }
640     else
641     {
642         circle_refine_radius = cfg.circle_rad;
643     }
644
645     LOG("refine radius: " << circle_refine_radius);
646
647
648     circle_delta[0] = delx / cfg.subdivs;
649     circle_delta[1] = dely / cfg.subdivs;
650
651     if (cfg.coarsen_thresh >=0)
652     {
653         circle_coarsen_thresh = cfg.coarsen_thresh;
654     }
655     LOG("coarsening threshold factor: " << circle_coarsen_thresh);
656
657     LOG("refinement interval: " << cfg.refine_interval);
658     LOG("coarsening interval: " << cfg.coarsen_interval);
659
660
661     if (cfg.fast_forward > 0)
662     {
663         LOG("fast-forwarding refinement front");
664         for (int f=0; f<cfg.fast_forward; f++)
665         {
666             if (f % cfg.refine_interval == 0)
667             {
668                 circle_center = circle_center + circle_delta;
669             }
670
671         }
672     }
673
674
675     // Begin refinement / coarsening circle propogation
676
677
678
679     for ( int current_circle_iteration = 0;
680           current_circle_iteration < cfg.num_iterations;
681           current_circle_iteration++ )
682     {
683         timer iteration_timer;
```

```
684
685         LOG_MEM(current_circle_iteration, "iter start");
686
687         using namespace splatter::query;
688         LOG("beginning big_circle iteration " << current_circle_iteration)
               ;
689
690         // coarsening
691         if (cfg.do_coarsen && current_circle_iteration % cfg.
               coarsen_interval == 0)
692         {
693             std::deque<int> to_coarsen;
694
695
696             // mark entities for de-refinement
697
698             mgr.query(
699                 for_each(coarsen_candidates(coarsenk)) >>=
700
701                 where(outside_radius(circle_center,
702                                      (1.0-circle_coarsen_thresh)*
                                         circle_refine_radius,
703                                      (1.0+circle_coarsen_thresh)*
                                         circle_refine_radius,
704                                      coords)) >>=
705
706                 save_ids(to_coarsen) >>=
707                 end()
708                 );
709
710             TBL(current_circle_iteration, "chgcoarsen", to_coarsen.size())
                   ;
711             LOG("CHANGE: coarsening " << to_coarsen.size() << "/" << mgr.
                   pctx().reduce((int)to_coarsen.size(), MPI_SUM));
712
713             part_timer.start();
714
715             // **** DE-REFINE THEM ****
716             do_coarsen(coarsenk, to_coarsen);
717
718             LOG("TIME: coarsening took " << part_timer.time());
719             TBL(current_circle_iteration, "*tmcoarsen", part_timer.time())
                   ;
720         }
721
722
723         LOG_MEM(current_circle_iteration, "iter  mid");
724
725         // refinement
726         if (current_circle_iteration % cfg.refine_interval == 0)
727         {
```

```
728             circle_center = circle_center + circle_delta;
729
730
731             std::deque<int> to_refine;
732
733             {
734                 // mark entities for refinement
735
736                 mgr.query(
737                     all_faces("tets") >>=
738
739                     // this test marks tet_ref_status
740                     where(on_radius(circle_center, circle_refine_radius,
                           coords)) >>=
741                     where(negate(flag(tet_is_refined, 1))) >>=
742                     save_ids(to_refine) >>= end()
743                     );
744             }
745
746
747             TBL(current_circle_iteration, "chgrefine", to_refine.size());
748             LOG("CHANGE: refining " << to_refine.size() << "/" << mgr.pctx
                   ().reduce((int)to_refine.size(), MPI_SUM));
749             part_timer.start();
750
751             // **** REFINE THEM ****
752             mgr.refine_marked("tets", to_refine, coarsenk, mgr.get_index("
                   bnds"));
753
754             LOG("TIME: refining took " << part_timer.time());
755             TBL(current_circle_iteration, "*tmrefine", part_timer.time());
756         }
757
758         LOG_MEM(current_circle_iteration, "iter postref");
759
760         // purge remote data for loadbalance
761         mgr.clear_phantom_data();
762         mgr.remove_unused_nodes();
763         int edgecut;
764
765         if (cfg.do_balance)
766         {
767
768             std::vector<int> new_numbering;
769             std::vector<int> new_node_dist;
770
771             // call ParMETIS in adaptation mode
772
773             LLOG(0,"decomp'ing to adapt");
774             part_timer.start();
775             decomp(mgr.pctx().comm(), mgr.get_index("tets"),
```

```
776                     mgr.pctx().np(), &edgecut,
777                     mgr.get_ownerdb(),
778                     new_numbering, new_node_dist, true,
779                     current_circle_iteration); // true->adapt
780             LOG("TIME: decomp (for load-balancing) took  " << part_timer.
                    time());
781             TBL(current_circle_iteration, "*decomp", part_timer.time());
782
783
784             LLOG(0, "renumbering to balance load");
785             part_timer.start();
786
787             // apply the new numbering and distribution
788
789             mgr.renumber(new_numbering, new_node_dist);
790             LOG("TIME: renumber for load balance took " << part_timer.time
                    ());
791             TBL(current_circle_iteration, "*renumber", part_timer.time());
792         }
793
794         // rebuild phantom data
795         mgr.augment_nodes();
796
797
798         // save restart files if required
799         if (cfg.do_save || (current_circle_iteration == cfg.num_iterations
                -1 && cfg.do_final_save))
800         {
801             // save restart file
802             outfile = cfg.save_name + "iter-" + stringify(
                    current_circle_iteration);
803             mgr.save_start(outfile);
804             mgr.save_node_data("coords");
805             // mgr.save_data("tets", "refstat");
806             mgr.save_end();
807         }
808
809         double mintet;
810
811
812         LOG("end of iteration " << current_circle_iteration << "; total
                volume: " <<
813             total_tet_volume(mgr, coords, &mintet) << " | total area: " <<
                    total_tri_area(mgr, coords));
814         LOG("minimum tet volume: " << mgr.pctx().reduce(mintet, MPI_MIN));
815         LOG("TIME: iteration " << current_circle_iteration << " took " <<
                iteration_timer.time());
816         TBL(current_circle_iteration, "*itertime", iteration_timer.time())
                ;
817
818         LOG("SIZE: nodes: " << mgr.get_ownerdb());
```

```
819         LOG("SIZE: tets: " << tet_idx->size() << "/" << mgr.pctx().reduce
               ((int)tet_idx->size(), MPI_SUM));
820         LOG("SIZE: bnds: " << bnd_idx->size() << "/" << mgr.pctx().reduce
               ((int)bnd_idx->size(), MPI_SUM));
821
822         TBL(current_circle_iteration, "nlocal_iter", mgr.get_ownerdb().
               nlocal());
823         TBL(current_circle_iteration, "coords_iter", coords.size());
824         TBL(current_circle_iteration, "tets_iter", (tets.size()/4));
825         TBL(current_circle_iteration, "bnds_iter", (bnds.size()/3));
826         LOG_MEM(current_circle_iteration, "iter final");
827
828     }
829
830     LOG("the end");
831     MPI_Finalize();
832 }
833
834
835
836 // refinement hooks
837 void new_node_coords(part_mgr* mgr, void* edgesp, void* udata)
838 {
839     const ownerdb& odb = mgr->get_ownerdb();
840     std::vector<starcd::node_coords>& coords = *(std::vector<starcd::
           node_coords>*)udata;
841
842     temp_index& edge_map=*((temp_index*)(edgesp));
843     const std::vector<int>& nmap = edge_map.data(refine_id_tag, std::
           vector<int>());
844
845     // for each refined edge, set the new node coordinate to the center of
            the
846     // edge
847
848     for (int e=0; e<edge_map.size(); e++)
849     {
850         if (odb.owns(nmap[e]))
851         {
852             int nn = odb.g2l(nmap[e]);
853             int n1 = odb.g2l_p(edge_map[e][0]);
854             int n2 = odb.g2l_p(edge_map[e][1]);
855
856             coords[nn] = (coords[n1] + coords[n2]) * 0.5;
857         }
858     }
859 }
860
861 void new_tet_cond(part_mgr* mgr, void* cbdata, void* udata)
862 {
```

```
863     std::vector<starcd::tet_cond_data>& vcond =*((std::vector<starcd::
            tet_cond_data>*)udata);
864
865     // propogate volume condition and tet_is_refined into new replacement
            tets
866
867     refine_arg* arg = (refine_arg*)cbdata;
868     for (int n=0; n<arg->numnew; n++)
869     {
870         vcond[arg->newent[n]] = vcond[arg->oldent];
871         tet_is_refined[arg->newent[n]]=1;
872     }
873 }
874
875
876
877
878
879 // get extent of mesh to calculate reasonable sphere radius for testing
880 void get_bounds(const std::vector<starcd::node_coords>& nc, const
        parallel_ctx& ctx,
881                 double* lowxp, double* hixp,
882                 double* lowyp, double* hiyp,
883                 double* lowzp, double* hizp)
884
885 {
886     double lowx = nc[0][0];
887     double hix = nc[0][0];
888     double lowy = nc[0][1];
889     double hiy = nc[0][1];
890     double lowz = nc[0][2];
891     double hiz = nc[0][2];
892
893     for (unsigned int n=1; n<nc.size(); n++)
894     {
895         double x = nc[n][0];
896         if (x < lowx) lowx=x;
897         if (x > hix) hix=x;
898
899         x = nc[n][1];
900         if (x < lowy) lowy=x;
901         if (x > hiy) hiy=x;
902
903          x = nc[n][2];
904         if (x < lowz) lowz=x;
905         if (x > hiz) hiz=x;
906     }
907
908     *lowxp = ctx.reduce(lowx, MPI_MIN);
909     *hixp = ctx.reduce(hix, MPI_MAX);
910     *lowyp = ctx.reduce(lowy, MPI_MIN);
```

```
911     *hiyp = ctx.reduce(hiy, MPI_MAX);
912     *lowzp = ctx.reduce(lowz, MPI_MIN);
913     *hizp = ctx.reduce(hiz, MPI_MAX);
914 
915     return;
916 }
917 
918 
919 
920 // run this after MPI_Init
921 status configure_run(runcfg& cfg, int argc, char** argv, int numload)
922 {
923     bool usage_err = false;
924     int ch;
925 
926     // defaults
927     cfg.do_load=false;
928     cfg.do_restart=false;
929     cfg.do_save=false;
930     cfg.init_decomp=true;
931     cfg.do_init_save=false;
932     cfg.do_final_save=false;
933     cfg.project="trial";
934     cfg.save_name="";
935     cfg.do_coarsen=false;
936     cfg.num_iterations=0;
937     cfg.num_loaders=numload;
938     cfg.do_balance=true;
939     cfg.coarsen_interval=1;
940     cfg.refine_interval=1;
941     cfg.subdivs=-1;
942     cfg.circle_rad=-1;
943     cfg.coarsen_thresh=-1;
944     cfg.fast_forward=0;
945 
946 
947     while ((ch = getopt(argc, argv, "l:p:ci:r:bo:C:R:v:?f:d:t:SFB")) !=
            -1)
948     {
949         switch (ch)
950         {
951          case 'l':
952              cfg.num_loaders = atoi(optarg);
953              if (cfg.num_loaders <= 0) usage_err=true;
954              break;
955 
956          case 'p':
957              cfg.do_load=true;
958              cfg.project=std::string(optarg);
959              break;
960 
```

```
961             case 'c':
962                 cfg.do_coarsen=true;
963                 break;
964 
965             case 'i':
966                 cfg.num_iterations = atoi(optarg);
967                 if (cfg.num_iterations < 0) usage_err=true;
968                 break;
969 
970             case 'r':
971                 cfg.do_restart=true;
972                 cfg.project=std::string(optarg);
973                 break;
974 
975             case 'b':
976                 cfg.do_balance=false;
977                 break;
978 
979             case 'o':
980                 cfg.do_save=true;
981                 cfg.save_name=std::string(optarg)+"-";
982                 break;
983 
984             case 'S':
985                 cfg.do_init_save=true;
986                 break;
987 
988             case 'F':
989                 cfg.do_final_save=true;
990                 break;
991 
992             case 'C':
993                 cfg.coarsen_interval = atoi(optarg);
994                 if (cfg.coarsen_interval <= 0) usage_err=true;
995                 break;
996 
997             case 'R':
998                 cfg.refine_interval = atoi(optarg);
999                 if (cfg.refine_interval <= 0) usage_err=true;
1000                break;
1001 
1002            case 'v':
1003                cfg.subdivs = atoi(optarg);
1004                if (cfg.subdivs <= 0) usage_err=true;
1005                break;
1006 
1007            case 'd':
1008                cfg.circle_rad = atof(optarg);
1009                if (cfg.circle_rad < 1E-6) usage_err=true;
1010                break;
1011
```

```
1012            case 't':
1013                cfg.coarsen_thresh = atof(optarg);
1014                if (cfg.coarsen_thresh < 1E-6) usage_err=true;
1015                break;
1016
1017            case 'B':
1018                cfg.init_decomp=false;
1019                break;
1020
1021
1022            case '?':
1023                std::cout << usage_str << std::cerr;
1024                MPI_Finalize();
1025                exit(0);
1026
1027
1028            case 'f':
1029                cfg.fast_forward = atoi(optarg);
1030                if (cfg.fast_forward < 0) usage_err=true;
1031                break;
1032
1033
1034            default:
1035                usage_err=true;
1036           }
1037       }
1038
1039       if (usage_err == true || (cfg.do_load && cfg.do_restart))
1040       {
1041           return SPLATT_FAIL;
1042       }
1043
1044       if (cfg.subdivs == -1)
1045       {
1046           cfg.subdivs = cfg.num_iterations;
1047       }
1048
1049       if (! cfg.do_restart)
1050       {
1051           cfg.do_load=true;
1052       }
1053
1054       return SPLATT_OK;
1055
1056   }
```

APPENDIX B
TETRAHEDRA VALIDATION QUERY MODULE

```
1
2  // This custom query module is used to process a tet stream, checking the
3  // currently refined edges and verifying that the resulting discretization
4  // will leave usable smaller tets.  It outputs additional edges that must
       be
5  // refined for the configuration to be valid.
6
7  class validate_tet : public splatter::query::query_op
8  {
9  public:
10     // initialize with internal status vector and temp_index containing
11     // current refinement candidates
12
13     validate_tet(std::vector<int>& refstat, const temp_index& edges) :
14         refstat(refstat), edges(edges)
15     {}
16
17
18     // This method does the job of a query filter module
19
20     SQ_FILTER(idx, eltno, etype, nnodes, nodes, o)
21     {
22         int edge[] = { nodes[0], nodes[1],
23                        nodes[0], nodes[2],
24                        nodes[0], nodes[3],
25                        nodes[1], nodes[2],
26                        nodes[1], nodes[3],
27                        nodes[2], nodes[3] };
28
29         // 1 or 2 refined edges, or exactly 3 co-facial edges is ok,
30         // More than this requires full tet validation
31         // ------------------------------------------------
32
33
34
35         // Partition tet edge lists into refined and unrefined
36         std::vector<int> rpart(6);
37         int rcount=0; int end=5;
38         for (int e=0; e< 12; e+= 2)
39         {
40             if (edges.find(&edge[e]) == -1)
41             {
42                 // unrefined
43                 rpart[end--]=e;
44             }
45             else
46             {
47                 // refined
48                 rpart[rcount++]=e;
49             }
```

```
50          }
51
52          assert(rcount==end+1);
53          refstat[eltno] = rcount;
54
55
56          if (rcount == 1)        // refining one edge is always ok
57          {
58              return;
59          }
60
61
62          if (rcount == 2)        // refining two edges requires the third
63                                  // cofacial edge if it exists
64          {
65              /* add 3rd cofacial edge if possible */
66              int lastedge[2];
67
68              if (edge[rpart[0]] == edge[rpart[1]])
69              {
70                  lastedge[0] = edge[rpart[0]+1];
71                  lastedge[1] = edge[rpart[1]+1];
72
73                  // output new edge
74                  o(NULL, -1, topo::EDGE,2, lastedge);
75                  refstat[eltno]=3;
76                  return;
77              }
78              else if (edge[rpart[0]] == edge[rpart[1]+1])
79              {
80                  lastedge[0] = edge[rpart[0]+1];
81                  lastedge[1] = edge[rpart[1]];
82
83                  // output new edge
84                  o(NULL, -1, topo::EDGE,2, lastedge);
85                  refstat[eltno]=3;
86                  return;
87              }
88              else if (edge[rpart[0]+1] == edge[rpart[1]])
89              {
90                  lastedge[0] = edge[rpart[0]];
91                  lastedge[1] = edge[rpart[1]+1];
92
93                  // output new edge
94                  o(NULL, -1, topo::EDGE,2, lastedge);
95                  refstat[eltno]=3;
96                  return;
97              }
98              else if (edge[rpart[0]+1] == edge[rpart[1]+1])
99              {
100                 lastedge[0] = edge[rpart[0]];
```

```
101                 lastedge[1] = edge[rpart[1]];
102
103                 // output new edge
104                 o(NULL, -1, topo::EDGE,2, lastedge);
105                 refstat[eltno]=3;
106                 return;
107             } else
108             {
109                 // no cofacial edge, so we leave it at 2
110 //              TRACE(11, "not a cofacial edge...");
111                 return;
112             }
113         }
114
115
116         if (rcount == 3)
117         {
118             // make sure the three refined edges are cofacial
119             if (edgeloop(&edge[rpart[0]], &edge[rpart[1]], &edge[rpart
                    [2]]))
120             {
121                 return;
122             }
123
124             // otherwise we refine the whole tet
125         }
126
127         // refine the whole tet
128         // TRACE(11, "need to refine the whole tet! " << edgecount);
129
130         refstat[eltno] = MAX_TET_EDGES;
131
132         // output all previously unrefined edges, indicating that they
                need to
133         // be refined
134         for (unsigned int e=rcount; e<6; e++)
135         {
136             o(NULL, -1, topo::EDGE, 2, &edge[rpart[e]]);
137         }
138
139         return;
140     }
141
142 private:
143     std::vector<int>& refstat;
144     const temp_index& edges;
145 }
```

APPENDIX C
API DOCUMENTATION

Introduction

The following pages describe the public methods for the major classes of the Splatter framework.

Data Structures

C.1 splatter::part_mgr Class Reference

C.1.1 Detailed Description

`part_mgr` is the main access point for users.

It is used to create and manipulate indices, as well as provide global mesh operations such as `renumber()`.

It is responsible for maintaining all local content (currently based on MPI's reported rank) and provides user access to relevant information via the `ownerdb`.

It maintains the overall topological mapping between indices, and ensures that index modifications are propogated properly – as long as they occur through the proper channels.

C.1.2 Constructor & Destructor Documentation

C.1.2.1 part_mgr::part_mgr (int *num_ent*, const **entity_cfg** * *cfg*, bool *doinit = true*)

Construct a new `part_mgr`

Parameters

num_ent	number of entities in cfg
cfg	pointer to an appropriate topology configuration
doinit	true if the part_mgr should immediately initialize usign the default parallel_ctx

C.1.2.2 splatter::part_mgr::part_mgr ( )

Construct a new part_mgr using topo::std_entities. Requires an explicit call to init()

C.1.3 Member Function Documentation

C.1.3.1 status part_mgr::init (**parallel_ctx** *pctx* = **parallel_ctx**())

Initialize this part_mgr with the specified parallel_ctx. Default to an auto-configured parallel_ctx based on MPI_COMM_WORLD

C.1.3.2 status part_mgr::finalize_load ( )

Make sure initial distribution of all indices is correct. Call only after calling part_mgr-::config_node_index and adding any other desired indices

C.1.3.3 **splatter::implicit_index** * part_mgr::config_node_index (int *size*, int *node_type* = *0*)

Designate size and other behavior (via flags) of the primary node index.

Parameters

size	number of local nodes owned by this rank
node_type	index in topo_cfg for nodes
flags	extra parameters for index creation

Returns

handle to node index

C.1.3.4 splatter::explicit_index * part_mgr::add_explicit_index (std::string *name*, int *entity_type*, std::vector< int > & *nodes*, int *flags = 0*)

Add a new homogeneous explicit index

Parameters

name	tag for this set of faces
entity_type	index in topo_cfg for this face type
nodes	vector of node ids definining the faces for this index
flags	extra parameters for index creation

Returns

handle to created index

C.1.3.5 splatter::explicit_index * part_mgr::add_explicit_index (std::string *name*, int *entity_type*, int *flags = 0*)

Add a new homogeneous explicit index without loading nodes

Parameters

name	tag for this set of faces
entity_type	index in topo_cfg for this face type
flags	extra parameters for index creation

Returns

handle to created index

C.1.3.6 splatter::explicit_index * part_mgr::add_explicit_index_notype (std::string *name*, int *width*, int *flags = 0*)

Add a new homogeneous explicit index without loading nodes or specifying an entity type

Parameters

name	tag for this set of faces
width	number of nodes per entity
flags	extra parameters for index creation

Returns

handle to created index

C.1.3.7 splatter::implicit_index * splatter::part_mgr::get_node_index ( )

Get the index managing node data

Returns

null if index is not configured

C.1.3.8 **splatter::explicit_index** * part_mgr::get_index (std::string *name*)

Get a named index

Parameters

name	name (tag) of desired index

Returns

null if index cannot be found

C.1.3.9 status part_mgr::track (temp_index * *idx*)

Track a temp_index. A tracked index provides phantom node dependencies and is renumbered

Parameters

idx	pointer to temp_index that should be tracked

C.1.3.10 status part_mgr::untrack (temp_index * *idx*)

Untrack a temp_index

Parameters

idx	pointer to temp_index that should be untracked

C.1.3.11 const parallel_ctx & splatter::part_mgr::pctx ( ) const

Access of this mesh's parallel_ctx

Returns

a const reference to this part_mgr's parallel_ctx

C.1.3.12 const ownerdb & splatter::part_mgr::get_ownerdb ( ) const

Access the authority of node/entity ownership for this mesh

Returns

a const reference to this part_mgr's ownerdb

C.1.3.13 **ownerdb** splatter::part_mgr::make_ownerdb (std::vector< int > *nd*)

Create a new ownerdb out of a user-specified node distribution

Parameters

nd	the node distribution to use – the user is responsible for making sure it works with the current parallel_ctx

Returns

the new ownerdb (by value)

C.1.3.14 const **entity_cfg** * splatter::part_mgr::get_etypes ( ) const

Get current array of valid entity types

Returns

a const pointer to the topological configuration

C.1.3.15 status part_mgr::augment_nodes (std::list< **temp_index** * > *extras = std::list*<**temp_index***>*()*)

Collect all phantom nodes from indices flagged SPLATT_PHANTOM_DEPS and those that are explicitly tracked and those passed in the parameter to this method.

Parameters

extras	explicit list of temp_index* from which to extract phantom nodes

C.1.3.16 status part_mgr::sync_all_node_data ( )

Update data associated with all phantom nodes

Returns

success or failure

C.1.3.17 status part_mgr::sync_node_data (std::string *tag*)

Update data named tag associated withphantom nodes

Returns

success or failure

C.1.3.18 template<typename T > status splatter::part_mgr::sync_node_data (std::vector< T > & *dat*, int *size_per = 1*)

Sync unattached data, assuming normal index/data relationship with nodes

Parameters

dat	node data to sync
size_per	data count per node

Returns

success or failure

C.1.3.19 template<typename T > status splatter::part_mgr::sync_raw_node_data (T * *dat*, int *size_per = 1*)

Sync unattached data, assuming normal index/data relationship with nodes

Parameters

dat	node data to sync
size_per	data count per node

Returns

success or failure

C.1.3.20 `status part_mgr::renumber ( const std::vector< int > & newgids, std::list< temp_index * > externals = std::list<temp_index*>() )`

Renumber the mesh, and optionally any `temp_index`'s specified

Parameters

newgids	vector of new global ids for the nodes owned by this rank
externals	list of temp_index* to also renumber

C.1.3.21 `status part_mgr::renumber ( const std::vector< int > & newgids, const std::vector< int > & dist, std::list< temp_index * > externals = std::list<temp_index*>() )`

Renumber and redistribute the mesh, and optionally any `temp_index`'s specified

Parameters

newgids	vector of new global ids for the nodes owned by this rank

dist	new node distribution array
externals	list of temp_index* to also renumber

C.1.3.22 `status part_mgr::remove_unused_nodes ( std::list< temp_index * > externals = std::list<temp_index*>() )`

Purge all nodes that do not appear in any index

Parameters

externals	list of temp_index* to check for used nodes

Returns

success or failure

C.1.3.23 `status part_mgr::build_fast_indices ( )`

Pre-build search structures for all indices flagged `SPLATT_FAST_INDEX`

Returns

success or failure

C.1.3.24 `status part_mgr::build_phantom_map ( std::list< temp_index * > extras = std::list<temp_index*>() )`

Do not call

C.1.3.25 `status part_mgr::build_node_sync_map ( )`

Do not call

C.1.3.26 status part_mgr::clear_phantom_data ( )

Remove all phantom node maps and data

Returns

success or failure

C.1.3.27 status part_mgr::get_more_nodes (int *amt*, int * *newstart*, std::list< **temp_index** * > *externals = std::list*<**temp_index***>*()*)

Request the creation of more nodes

Parameters

amt	number of nodes needed on the local partition
newstart	output variable for new starting node id
externals	list of `temp_index` to renumber after getting more nodes

C.1.3.28 template<typename T > status splatter::part_mgr::query (T *op*)

Querying the mesh

Parameters

op	the query

Returns

success or failure

C.1.3.29 status part_mgr::set_query_data (std::string *name*, std::deque< int > * *data*)

do not call (reserved for query usage)

C.1.3.30 std::deque< int > * part_mgr::get_query_data (std::string *name*)

do not call (reserved for query usage)

C.1.3.31 status part_mgr::register_hook (hook_entry *point*, int *entity*, **app_hook** * *hook*)

Register callback for when certain action occurs

Parameters

point	event to trigger callback (e.g., SPLATT_RENUMBER_HOOK or SPLATT_REFINE_HOOK)
entity	type of entity for this hook to process, if applicable
hook	the callback function

Returns

success or failure

C.1.3.32 void part_mgr::dump ( )

Print out various information about the state of the mesh

C.1.3.33 `status part_mgr::refine_marked ( std::string` *indexname,*
`std::deque< int > &` *marked,* **explicit_index** `*` *next* `= NULL )`

Refine the contents of index with specified positions

Parameters

indexname	name of index to refine
marked	collection of entity ids to refine
next	secondary index affected by refinement

Returns

success or failure

C.1.3.34 `status part_mgr::refine_marked ( std::string index_name, std::deque< int > & marked, coarsen_kernel * kernel, explicit_index * next = NULL )`

Refine the contents of index with specified positions, allowing de-refinement

Parameters

indexname	name of index to refine
marked	collection of entity ids to refine
kernel	handle on derefinement bookkeeping
next	secondary index affected by refinement

Returns

success or failure

C.1.3.35 `status part_mgr::refine_marked_opt ( std::string indexname, std::deque< int > & marked, temp_index & edgemap, std::vector< int > & new_ids, std::vector< int > & unused_nodes, explicit_index * next = NULL )`

Internal refinement. **Do not call**

C.1.3.36 void splatter::part_mgr::inhibit_hooks ( )

Supress all hooks

C.1.3.37 void splatter::part_mgr::enable_hooks ( )

Reactivate all hooks

C.1.3.38 void part_mgr::call_hooks ( )

Explicitly trigger all renumber hooks

C.1.3.39 status part_mgr::save_start (std::string *path_prefix*)

Initiate creation of restart files

Parameters

path_prefix	location of restart files

Returns

success or failure

C.1.3.40 status part_mgr::save_node_data (std::string *tag*)

Save node data in current restart files

Parameters

tag	node data tag to save

Returns

success or failure

C.1.3.41 status part_mgr::save_data (std::string *idx*, std::string *tag*)

Save index data in current restart files

Parameters

idx	index whose data should be saved
tag	data tag to save

Returns

success or failure

C.1.3.42 status part_mgr::save_end ( )

Finish creation of all restart files

Returns

success or failure

C.1.3.43 int part_mgr::load (std::string *path_prefix*)

Restore `part_mgr` state from restart files

Returns

number of nodes loaded

C.2.1 Detailed Description

Container of all MPI details for the current mesh

C.2.2 Constructor & Destructor Documentation

C.2.2.1 parallel_ctx::parallel_ctx (MPI_Comm *comm*) [explicit]

Instantiate with provided MPI communicator

Parameters

comm	the MPI communicator to use

C.2.2.2 parallel_ctx::parallel_ctx ( ) [explicit]

Instantiate placeholder context. Requires `init()` prior to use

C.2.2.3 parallel_ctx::parallel_ctx (const **parallel_ctx** & *other*)

Create new `parallel_ctx` using values from another

Parameters

other	the other `parallel_ctx` to copy

C.2.3 Member Function Documentation

C.2.3.1 void parallel_ctx::init (MPI_Comm *comm*)

Initialize this `parallel_ctx`

Parameters

comm	the MPI communicator to use

C.2.3.2 void parallel_ctx::rebind_stdio (int *ignore_rank = -1*, std::string *file_prefix = ""*) const

Rebind stdout/stderr to file stdout.<rank>

Parameters

ignore_rank	MPI rank that should continue printing to stdout
file_prefix	optional prefix before stdout/stderr in filename

C.2.3.3 int splatter::parallel_ctx::np ( ) const

Returns

number of processors in this MPI communicator

C.2.3.4 int splatter::parallel_ctx::rank ( ) const

Returns

this process's rank on this MPI communicator

C.2.3.5 MPI_Comm& splatter::parallel_ctx::comm ( ) const

Returns

this MPI communicator

C.2.3.6 int splatter::parallel_ctx::root ( ) const

Returns

rank of processor considered the official source in broadcast

C.2.3.7 template<typename T > T splatter::parallel_ctx::reduce (T *in*, MPI_Op *op*) const

Wrapper around MPI_Reduce

Parameters

in	local data for reduction
op	MPI operator for reduction

Returns

global reduction value

C.2.3.8 template<typename T > void splatter::parallel_ctx::broadcast (std::vector< T > & *vec*, int *root = -1*) const

Wrapper around MPI_Broadcast

Parameters

vec	data to broadcast
root	source of official data (see root() if -1 is used

C.2.3.9 void parallel_ctx::barrier ( ) const

Wrapper around MPI_Barrier

C.2.3.10 **parallel_ctx** & parallel_ctx::operator= (const **parallel_ctx** & *other*)

Assignment operator

Parameters

other	the source of the assigned values

C.3 splatter::ownerdb Class Reference

C.3.1 Detailed Description

Container of all parallel distribution / node ownership data

C.3.2 Constructor & Destructor Documentation

C.3.2.1 `splatter::ownerdb::ownerdb ( )`

Public constructor: creates invalid `ownerdb`

C.3.3 Member Function Documentation

C.3.3.1 `bool splatter::ownerdb::owns ( int g ) const`

Parameters

g	the global node id being considered

Returns

true iff the local processor owns global node g

C.3.3.2 `int splatter::ownerdb::owner ( int g ) const`

Parameters

g	the global node id being considered

Returns

the rank of hte process that owns global node g

C.3.3.3 int splatter::ownerdb::me ( ) const

Returns

the local processor's rank

C.3.3.4 int splatter::ownerdb::np ( ) const

Returns

the number of processors in this MPI communicator

C.3.3.5 int splatter::ownerdb::high ( ) const

Returns

the high node id (exclusive) assigned to the local rank

C.3.3.6 int splatter::ownerdb::low ( ) const

Returns

the low node id (inclusive) assigned to the local rank

C.3.3.7 int splatter::ownerdb::g2l (int *g*) const

Parameters

g	the global node id being considered

Returns

the corresponding local node id, assuming it is owned locally

C.3.3.8 int splatter::ownerdb::l2g (int *l*) const

Parameters

l	the local node id being considered

Returns

the corresponding global node id, assuming l is valid

C.3.3.9 int splatter::ownerdb::g2l_p (int *g*) const

Parameters

g	the global node id being considered

Returns

the corresponding local node id, assuming it is owned locally or a phantom node

C.3.3.10 int splatter::ownerdb::l2g_p (int *l*) const

Parameters

l	the local node id being considered

Returns

the corresponding local node id, assuming it is owned locally or a phantom node

C.3.3.11 int splatter::ownerdb::nlocal ( ) const

Returns

the number of nodes owned locally

C.3.3.12 int splatter::ownerdb::nglobal ( ) const

Returns

the number of global nodes

C.3.3.13 int splatter::ownerdb::nlocal_p ( ) const

Returns

the total number of local nodes, including phantom nodes

C.3.3.14 int splatter::ownerdb::nphantom ( ) const

Returns

the number of phantom nodes locally

C.3.3.15 `const int * splatter::ownerdb::node_dist ( ) const`

Returns

the underlying node distribution

C.3.3.16 `const` **parallel_ctx** `& splatter::ownerdb::pctx ( ) const`

Returns

the underlying `parallel_ctx`

C.3.3.17 `void ownerdb::debug_phantoms ( ) const`

Do not call

C.4 splatter::index_op Class Reference

C.4.1 Detailed Description

Functionality required for use in `index::apply`

Any functor used in apply must have something with this operator (It is not required that this class be explicitly extended)

C.4.2 Member Function Documentation

C.4.2.1 virtual void splatter::index_op::operator() (**splatter::index** * *idx*, int *eltno*, int *etype*, int *nnodes*, int * *nodes*) const [pure virtual]

This operator will be called on each member of the index to which it's applied. Due to the way it's done with templates, it can be optimized into direct manipulation of the index – Compile-time polymorphism!

Parameters

idx	index being applied to
eltno	id of current entity
etype	type of current entity
nnodes	number of nodes in current entity
nodes	actual nodes of current entity

C.5 splatter::index Class Reference

C.5.1 Detailed Description

Superclass for other indices; collection of mesh entities.

C.5.2 Constructor & Destructor Documentation

C.5.2.1 `index::index ( const` **ownerdb** `& odb, std::string name,` **index_type** `type, int flags ) [protected]`

Superclass constructor for all index instances.

Parameters

reference	to ownership authority
name	identifying tag for this index
type	index_type for this index (for dispatching non-virtual template functions)

flags	index creation flags

C.5.3 Member Function Documentation

C.5.3.1 `status index::attach_data ( std::string` *name,* **data_proxy** $*$ *p* `)`

Associate data from a `data_proxy` with this index. Changes to the index (through the appropriate means) will be mirrored in the associated data.

NOTE: index destructor will delete the proxy.

Parameters

name	identifying tag for the attached data
p	pointer to proxy

Returns

success or failure

C.5.3.2 `status index::detach_data ( std::string` *name* `)`

Remove associated data, deleting proxy.

C.5.3.3 `template<typename D > D & splatter::index::data ( std::string` *name,* `const D &` *orig* `)`

Access associated data.

Parameters

name	identifying tag of requested data
orig	model data type (it's complicated...)

C.5.3.4 `std::string splatter::index::name ( ) const`

Returns

identifying name used for this index (used in mesh queries)

C.5.3.5 **index_type** `splatter::index::type ( ) const`

Returns

type of actual index value

C.5.3.6 `int splatter::index::flags ( ) const`

Returns

flags used to configure the index

C.5.3.7 `int splatter::index::size ( ) const [virtual]`

Returns

the number of entities in the index

Reimplemented in **splatter::explicit_index** , and **splatter::implicit_index** .

C.5.3.8 template<typename OP , typename IT > void splatter::index::apply (const OP & *op,* IT *start,* IT *end*)

Apply op to each entity in this index, with the positional id's specified by the iterator

Parameters

op	see `index_op`
start	iterator to beginning face id
end	end iterator

C.5.3.9 `template<typename OP > void splatter::index::apply ( const OP & op )`

Apply op to every entityt in this index

Parameters

op	see `index_op`

C.5.3.10 `template<typename OP , typename IT , void(OP::*)(index *, int, int, int, int *) const func> void splatter::index::apply ( const OP & op, IT start, IT end )`

Apply some `index_op` - esque member function to each entity in this index, with the positional id's specified by the iterator

Parameters

op	see `index_op`
start	iterator to beginning face id
end	end iterator

C.5.3.11 `template<typename OP , void(OP::*)(index *, int, int, int, int *) const func> void splatter::index::apply ( const OP & op )`

Apply some `index_op` - esque member function to every entity in this index

Parameters

op	see index_op

C.5.3.12 `virtual status splatter::index::renumber ( const` **ownerdb** `& new_odb, const std::vector< int > & local_gid, std::map< int, int > & global_map ) [protected], [pure virtual]`

pure virtual method for renumbering entities according to local_gid, and potentially redistributing nodes according to new_odb

Implemented in **splatter::explicit_index** , and **splatter::temp_index** .

C.6 splatter::explicit_index Class Reference

C.6.1 Detailed Description

An index of homogeneous entities and associated data. Potentially (probably) hashes node positions for fast lookup.

C.6.2 Constructor & Destructor Documentation

C.6.2.1 `explicit_index::explicit_index ( const` **ownerdb** `& odb, std::string name, int width, int entity_type, std::vector< int > & nodes, int flags = 0 )`

Constructor: called by part_mgr

C.6.2.2 explicit_index::explicit_index (const **ownerdb** & *odb*, std::string *name*, int *width*, int *entity_type*, int *flags = 0*)

Constructor: called by part_mgr

C.6.3 Member Function Documentation

C.6.3.1 int explicit_index::size ( ) const [virtual]

Returns

number of entities stored in index

Reimplemented from **splatter::index** .

C.6.3.2 const int* splatter::explicit_index::operator[] (int *f*) const

unchecked (raw) entity lookup

Parameters

f	desired entity position

Returns

const int* to nodes belonging to entity at position f

C.6.3.3 status explicit_index::build_hash ( )

Enable fast lookup for this index

C.6.3.4 int explicit_index::add_face (const int * *newface_nodes*)

Add a single entity to this index – respects fast lookup hash if necessary NOTE: user must ensure that data is extended appropriately.

Parameters

newface_-nodes	nodes to use for the new entity

Returns

position of new entity

C.6.3.5 const intset& splatter::explicit_index::faces (int *g*) const

Figure out which faces contain a node – hash must be enabled

Parameters

g	global id of desired node

Returns

const reference to intset of related faces

C.6.3.6 int splatter::explicit_index::width ( ) const

Returns

number of nodes in this index's entity type

C.6.3.7 const std::vector<int>& splatter::explicit_index::nodes ( ) const

Returns

raw pointer to indexed nodes

C.6.3.8 `bool splatter::explicit_index::is_hashed ( )`

Returns

whether fast lookup hash is enabled for this index

C.6.3.9 `void explicit_index::set_purge ( bool` *p* `)`

Specify whether this index should delete local nodes when no longer needed

C.6.3.10 `const` **fast_index&** `splatter::explicit_index::hash ( )`

Returns

const reference to the fast lookup hash

C.6.3.11 `template<typename Q_OP > void explicit_index::do_query ( const Q_OP &` *op,* `int` *specific = -1* `)`

Used to apply a query to this index – has to be public, but don't call it directly

C.6.3.12 `template<typename Q_OP , typename IT > void explicit_index::do_query_it ( const Q_OP &` *op,* `IT` *begin,* `IT` *end* `)`

Used to apply a query to this index, faces pulled by iterator

C.6.3.13 **exp_index_searcher** `splatter::explicit_index::get_query_searcher ( )`

Do not call outside a query.

Returns

a query object for searching this index quickly

C.6.3.14 template<typename IT > status explicit_index::delete_faces (IT *start*, IT *end*)

Globally delete all faces with ids iterated over between start and end

Parameters

start	beginning iterator to container of entity ids to delete
end	end iterator of container

Returns

success or failure

C.6.3.15 status explicit_index::sync_phantom_layer (bool *clobber_duplicates = true*)

Globally delete all faces, not trusting the hash to be picky enough. Use this with no-type indices that may have repeated node-ids

Ensures that all entities are stored on processes that need them (based on ownership of underlying entity nodes).

Parameters

clobber_duplicates	check and purge local entities that are duplicates

C.6.3.16 `int splatter::explicit_index::etype ( ) const`

Returns

entity type for this index

C.6.3.17 `status explicit_index::validate_hash ( )`

Expensive debugging/validation method **Do not call.**

C.6.3.18 `status explicit_index::validate_entities ( )`

Expensive debugging/validation method **Do not call.**

C.6.3.19 `status explicit_index::do_migrate ( migrate_args & args ) [protected], [virtual]`

move faces according to `migrate_args` – used mostly for deleting?? dangerous!

Reimplemented in **splatter::temp_index** .

C.6.3.20 `status explicit_index::renumber ( const ownerdb & new_odb, const std::vector< int > & local_gid, std::map< int, int > & global_map ) [protected], [virtual]`

pure virtual method for renumbering entities according to local_gid, and potentially redistributing nodes according to new_odb

Implements **splatter::index** .

Reimplemented in **splatter::temp_index** .

C.7 splatter::implicit_index Class Reference

C.7.1 Detailed Description

index subclass for collections of single-node entities, stored as a range of node ids – specifically the range of nodes owned by the current rank and potentially any phantom entities

This is really just used for the master node index, but i guess it could conceivably be used for something else...

C.7.2 Constructor & Destructor Documentation

C.7.2.1 splatter::implicit_index::implicit_index (const ownerdb & *odb*)

Constructor: called by `part_mgr`

C.7.3 Member Function Documentation

C.7.3.1 int splatter::implicit_index::size ( ) const [virtual]

Returns

number of nodes represented by this index (just nlocal!)

Reimplemented from **splatter::index** .

C.8 splatter::temp_index Class Reference

C.8.1 Detailed Description

a special, user-space version of an `explicit_index` for facilitating complex mesh manipulation logic. Be careful! It tries to maintain a mapping to position in original index.

Fast hash lookup is always enabled.

C.8.2 Constructor & Destructor Documentation

C.8.2.1 `temp_index::temp_index ( part_mgr * mgr, int entity_type, bool update_hash = false )`

Construct a new `temp_index`

Parameters

mgr	pointer to relevant `part_mgr`
entity_type	topological entity type for the contents of this index
update_hash	will this entity have enough distribution of mesh elements for the hash to benefit from having dedicate room for data associated with every local node ?

C.8.2.2 `temp_index::temp_index ( explicit_index & ei, bool update_hash = false )`

Construct a new `temp_index` modeled after an explicit index

Parameters

ei	`explicit_index` to model after
update_hash	this index is expected to use a wide range of local nodes

C.8.3 Member Function Documentation

C.8.3.1 **temp_index** temp_index::no_type (**part_mgr** * *mgr*, int *width*, bool *update_hash* = *false*) [static]

Construct a new temp_index with no type

Parameters

mgr	the authority `part_mgr`
width	number of nodes per entity
update_hash	this index is expected to use a wide range of local nodes

C.8.3.2 **temp_index** `temp_index::vec_wrapper (` **part_mgr** `*` ***mgr,*** `std::vector< int > & data ) [static]`

Temporarily treat an array as a `temp_index` so it can be renumbered

Parameters

mgr	the authority `part_mgr`
data	the `std::vector` to process

C.8.3.3 `status temp_index::clear ( )`

Remove all index contents

C.8.3.4 `int* splatter::temp_index::operator[] ( int f )`

Non-const entity accessor.

Parameters

f	entity to look up

Returns

pointer to nodes of specified entity

C.8.3.5 const int* splatter::temp_index::operator[] (int *f*) const

Const version of entity accessor.

Parameters

f	entity to look up

Returns

pointer to nodes of specified entity

C.8.3.6 `int splatter::temp_index::add_face ( int` *`eltno`*`, const int *` *`newface_nodes`* `)`

Add a entity to this index

Parameters

eltno	position in source index for entity, if applicable
newface_-nodes	nodes for the new entity

Returns

local position of new entity

C.8.3.7 `int splatter::temp_index::add_face ( const int *` *`newface_nodes`* `)`

Add a entity to this index

Parameters

newface_-nodes	nodes for the new entity

Returns

local position of new entity

C.8.3.8 status splatter::temp_index::add_all (const **temp_index** & *other,*
bool *unique = false*)

Copy all entities from one temp_index

Parameters

other	the original `temp_index`
unique	check for and purge duplicates

Returns

success or failure

C.8.3.9 `int splatter::temp_index::origid ( int f ) const`

Parameters

f	local entity being checked

Returns

original id in source index

C.8.3.10 `int splatter::temp_index::find ( const int * nodes ) const`

Finds a face in this index

Parameters

nodes	the face to be searched for

Returns

position of face or -1 if not found

C.8.3.11 `status temp_index::renumber ( const` **ownerdb** `& new_odb, const std::vector< int > & newgids, std::map< int, int > & global_map ) [virtual]`

Renumber this index (see `part_mgr`)

Reimplemented from **splatter::explicit_index** .

C.8.3.12 `template<typename Q_OP > void splatter::temp_index::do_query ( const Q_OP & op, int specific = -1 )`

Query support **Do not call directly**

C.8.3.13 `status splatter::temp_index::do_migrate ( migrate_args & args ) [virtual]`

Apply the `migrate` protocol

Parameters

args	protocol args

Returns

success or failure

Reimplemented from **splatter::explicit_index** .

C.9 splatter::data_proxy Class Reference

C.9.1 Detailed Description

Proxy for user data

C.9.2 Member Function Documentation

C.9.2.1 virtual int splatter::data_proxy::size ( ) [pure virtual]

Returns

size of data being proxied

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.2 virtual int splatter::data_proxy::size_per ( ) [pure virtual]

Returns

number of data elements per entity

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.3 virtual void splatter::data_proxy::clear ( ) [pure virtual]

Removes all data contents (be careful!)

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.4 virtual void* splatter::data_proxy::raw ( ) [pure virtual]

Returns

pointer to underlying data (be careful!)

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.5 virtual status splatter::data_proxy::resize (unsigned int *s*) [pure virtual]

Resize underlying data

Parameters

s	number of elements to resize for

Returns

success or failure

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.6 virtual void splatter::data_proxy::insert (int *n*) [pure virtual]

Insert slots for new data

Parameters

n	number of slots to insert

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.7 virtual void splatter::data_proxy::add_one ( ) [virtual]

Insert 1 slot for new data

C.9.2.8 virtual status splatter::data_proxy::batch_migrate (const **migrate_args** & *args*, **proxy_monitor** * *m* = *NULL*) [pure virtual]

Apply the batch_migrate protocol.

Parameters

args	protocol args

Returns

success or failure

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.9 `virtual` **data_proxy*** `splatter::data_proxy::deliver (` `const` **deliver_args** `&` ***args,*** **data_proxy** `*` ***out*** `=` ***NULL*** `)` `[pure virtual]`

Apply the `deliver` protocol.

Parameters

args	protocol args
out	optional proxy for collected data

Returns

pointer to collecting proxy

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.10 `virtual status splatter::data_proxy::sync (` **parallel_ctx** ***pctx,*** `const` **sync_args** `&` ***args*** `)` `[pure virtual]`

Apply the `sync` protocol.

Parameters

pctx	the `parallel_ctx` to use
args	protocol args

Returns

success or failure

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

`C.9.2.11 virtual status splatter::data_proxy::reorder ( const std::vector< int > & localids ) [pure virtual]`

Apply the ‘reorder protocol.

Parameters

localids	new local ids for all local data

Returns

success or failure

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

`C.9.2.12 virtual status splatter::data_proxy::load ( std::ifstream & in ) [pure virtual]`

Load data from restart file stream.

Parameters

in	the restart file stream.

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.13 `virtual status splatter::data_proxy::save ( std::ofstream & out, int limit = -1 ) [pure virtual]`

Save data to a restart file stream.

Parameters

out	the restart file stream.

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.14 `virtual` **data_proxy*** `splatter::data_proxy::type_copy ( ) [pure virtual]`

Create a new `data_proxy` capable of storing the same kind of data

Returns

the type clone

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

C.9.2.15 `virtual void splatter::data_proxy::store_raw ( int dest, int src,` **data_proxy** `* src_proxy ) [pure virtual]`

`Forcibly copy data between proxies, assumign they are type compatible`

Parameters

dest	position in local data
src	position in source data
src_proxy	data proxy containing source data

Implemented in **splatter::data_proxy_std< T >** , and **splatter::data_proxy_std< int >** .

VITA

Degrees

M.S. Computer Science
Tulane University, 1998
Thesis: Navigation and map-making with a team of mobile robots

B.S.E. Computer Engineering
Tulane University, 1997
Minor: Robotics and Automation

Positions

Lecturer, *University of Tennessee at Chattanooga* 2010-

Research Assistant, *UTC SimCenter* 2007-2010

Consultant / Developer, *Tanis Tech LLC* 2006-2010

Senior Programmer, *Advance Internet* 1999-2006

Professional Associate, *Johns Hopkins Applied Physics Lab* 1998-1999

Research Assistant, *Tulane University* 1996-1998

Teaching Assistant, *Tulane University* 1997-1998

Publications

"Petrov-Galerkin and discontinuous-Galerkin methods for time-domain and frequency-domain electromagnetic simulations." W. K. Anderson, L. Wang, S. Kapadia, C. Tanis, and B. Hilbert. *Journal of Computational Physics, vol. 230, no. 23, Sep. 2011.*

"Distributed Map-making Using Online Generalized Voronoi Graphs." J. Jennings, C. Kirkwood-Watts, C. Tanis. *Proceedings of the Conference on Automated Learning and Discovery (CONALD 98).*

"Cooperative Localization and Map-making for Mobile Robots." C. Tanis *Tulane University Technical Report, May 1997.*