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

Generic Components for Petascale Adaptive Unstructured Mesh Simulations

Ting Xie · Seegyoung Seol · Mark S. Shephard

Received: date / Revised version: date

Abstract In the traditional programming paradigm, data structures and algorithms are developed for specific data types and requirements. This leads to code redundancy and inflexibility thus not allowing effective code reuse for similar applications. One effective approach to increase code reuse is generic programming, which focuses on the development of efficient, reusable software libraries through suitable abstractions for the common requirements. In this paper, we present how we applied generic programming to an on-going effort for mesh-based adaptive simulations on massively parallel computers. Three generic components, iterator, set and tag, were developed using design pattern, C++ template programming and the Standard Template Library (STL). The scaling studies on petascale supercomputers demonstrate the efficiency of the reusable, generic components which do not sacrifice the performance of the previous tools developed in the traditional object-oriented programming paradigm.

1 Introduction

The generic programming [1,2] paradigm has emerged as a methodology towards developing efficient, reusable component-based software libraries, and gained popularity through the success of the C++ *Standard Template Library* (STL) [1,3,4]. Generic programming has been applied to scientific computing software packages such as:

- Matrix Template Library (MTL): a generic component library for high performance numerical linear algebra [5],
- Generic Message Passing framework (GMP): message communication library [6],
- Grid Algorithms Library (GrAL): a generic grid toolbox for reusable mesh-level components [7,8],
- Computational Geometry Algorithms Library (CGAL): a library for general-purpose geometric data structures and algorithms [9],
- Generic interfaces for parallel and adaptive hierarchical grids based on DUNE [10,11],
- A layer of generic software components used for parallelization of a finite element solver and for solver coupling in multi-physics applications [12].

The software components for an adaptive analysis of partial differential equations (PDEs) include [13–15]: *(i)* the geometric model which houses the topological and shape description of the domain of the problem, *(ii)* attributes describing the rest of parameters needed to define and quantify the problem, *(iii)* the mesh which describes the discretized representation of the domain used by the analysis method, and *(iv)* fields which describe the distribution of solution tensors over the mesh entities.

The most common approach for developing reusable simulation software is to create libraries for specific data components such as mesh, geometric model, and field and to let them interact through well-defined API's to perform needed operations to accomplish scientific applications. The Interoperable Technologies for Advanced Petascale Simulations (ITAPS) center [16] has defined a set of common interfaces that support the abstract data model; five that correspond to the core data model components, Geometry (*iGeom*), Mesh (*iMesh/iMeshP*), Field (*iField*), and Data Relation Manager (*iRel*) and one that contains the utilities and definitions used by other interfaces (*iBase*) [16–18]. A common set

Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY 12180, U.S.A.
E-mail: seols@rpi.edu

of utilities used in the five functional components are iterator, set and tag. This paper presents a generic implementation of these three utilities that are designed to support mesh based simulations on massively parallel computers.

The rest of the paper is organized as follows. Section 2 introduces the abstract data models of a mesh-based simulation, and summarizes a set of generic components for adaptive mesh-based simulations. Sections 3 - 5 present the three generic components developed as part of utilities of simulation tools, and describe how these generic components are used in the simulation model and data management software development to support automated adaptive simulations. In Section 6, the performance results of adaptive simulations on massively parallel computers are presented. Section 7 summarizes the paper and discusses the future work.

Nomenclature

V	the model, $V \in \{G, P, M\}$ where G signifies the geometric model, P signifies the partition model, and M signifies the mesh model.
V_i^d	the i^{th} entity of dimension d in model V. $d = 0$ for a vertex, $d = 1$ for an edge, $d = 2$ for a face, and $d = 3$ for a region.
P_i	the i^{th} part in a distributed mesh.

2 Simulation Data Model

The four data models central to general numerical solutions of PDEs with mesh based methods are:

- *Geometric model:* geometric model interface that supports the ability to interrogate solid models for topological adjacency and geometric shape information.
- *Mesh:* distributed mesh representation.
- *Field:* representation of tensor fields to quantify the distribution of physical parameters over mesh entities.
- *Relationship manager:* utility used to manage the relationships between meshes and geometric models, tensor fields and meshes, and so on.

The design of reusable and efficient generic components with appropriate concepts for mesh-based simulations is availed from understanding the common requirements of essential data models for parallel mesh-based simulations as well as a suitable level of abstractions in the form of *concepts* [1] of such requirements with ultimate balance between commonality and specialization.

In addition to the four data models described above, adaptive simulations in a parallel computing environment place extra demands to represent and manipulate the distributed mesh data over a large number of processing cores (i.e. *processors*). The subsections that follow present the data models and functional requirements of the geometric model, mesh, and distributed mesh as a first step towards identifying essential, reusable generic components.

2.1 Geometric Model

The geometric model is a data model which provides a functional interface to support the communication of geometry information to mesh-based applications.

Geometric entities: the constituents of a geometric model. They are, in the boundary representation, *regions, shells, faces, loops, edges* and *vertices* and *use* entities for vertices, edges, loops and faces with a non-manifold model [19].

Adjacencies: how geometric entities are connected to each other.

Geometric interrogations: provide specific information relating to the shape of geometric entities such as pointwise locations and shape coefficients [20–22].

Geometric entity sets: mechanism to group geometric entities for various purposes [16,23]. The useful attributes and requirements of geometric entity sets are: *(i)* entity uniqueness and entity insertion order preservation *(ii)* set population through entity addition or entity removal *(iii)* traversal through an iterator per entity type *(iv)* set binary operations (union, subtraction, intersection) *(v)* relationships among entity sets such as superset/subset and parent/child.

Tags: mechanism to attach arbitrary user data, termed as *tag data*, to geometric entity sets or geometric entities [16,23]. Tag data is a single or array of a specific type where it could be of primary data type such as integer, double, geometric entity set, geometric entity, or arbitrary type represented as *void*$*$.

Iterators: mechanism to traverse geometric entities by type either in an entity set or in the entire model [16].

2.2 Mesh

The mesh is a data model which provides a description of the mesh information in a manner that mesh-based operations can be efficiently performed. A minimum set of functional requirements to support adaptive simulations includes:

Mesh entities: the constituents of a mesh. They are distinguished by their type, i.e. topological dimension[1], (*vertex* (0D), *edge* (1D), *face* (2D), or *region* (3D)), and topology (for instance, triangle and quadrilateral for 2-dimensional entities, or tetrahedron or hexahedron for 3-dimensional entities) [24].

Adjacencies: how the topological mesh entities are connected to each other. For an entity of dimension d, first-order adjacency returns the mesh entities of dimension q, which are either on the closure of the entity for a downward adjacency ($d > q$), or for which the entity is part of the closure for an upward adjacency ($d < q$) [17,25–28].

Geometric classification: a unique association that each mesh entity maintains to a geometric entity for partial representation. Given a geometric entity, the set of equal dimension mesh entities classified on the geometric entity is termed as the *reverse classification* for the geometric entity [25].

Mesh entity sets: mechanism to group mesh entities for various purposes [18,23]. There are various kinds of sets depending on the following options:
- entity uniqueness
- entity insertion order preservation
- entity type constraint

In addition, *(i)* set population through entity addition or entity removal *(ii)* traversal through an iterator per entity type and/or topology *(iii)* set binary operations such as subtraction, intersection, and union *(iv)* relationships among sets such as superset/subset and parent/child are needed for flexible set manipulation.

Tags: mechanism to attach arbitrary user data, termed as *tag data*, to a part, entity set or mesh entity [18, 23]. Tag data is a single or array of a specific type where it could be of primary data type such as integer, double, mesh entity set and mesh entity, or arbitrary type represented as *void∗*.

Iterators: mechanism to traverse mesh entities in a specific range with various options [16,18], such as *(i)* traversing entities by type and/or topology *(ii)* traversing entities classified on a specific geometric entity, and so on.

2.3 Distributed Mesh

In parallel adaptive simulations, a mesh is distributed to parts over multiple processors. A distributed mesh data structure supports a topological representation of the distributed mesh and efficient distributed mesh manipulation functions. A topological model for a distributed mesh representation, referred to as a *partition model*, will support the needed capabilities for representing parts, part boundaries, mesh migration and partitioning [27,28].

[1] In this paper, entity *dimension* and entity *type* are interchangeable terms.

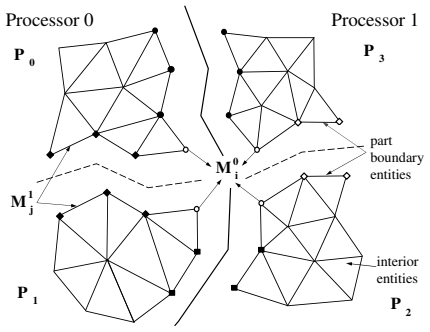

Fig. 1 Distributed 2D mesh on two processors with two parts per processor.

2.3.1 Part

When a mesh is distributed to N parts, each part is assigned to a processor. A part is a subset of topological mesh entities of the entire mesh, uniquely identified by its part handle or id, denoted by P_i, $0 \leq i < N$. Figure 1 depicts a 2D mesh that is distributed to four parts on two processors where each processor contains two parts. The dashed lines represent intra-processor part boundaries within a processor and the solid black lines represent inter-processor part boundaries between processors.

In practice, for effective manipulation of multiple parts per processor, a *mesh instance* is defined on each processor to contain the parts on that processor.

Note a mesh entity set can contain any mesh entities regardless of the belonging parts. In a distributed mesh environment, there are two types of mesh entity sets depending on the number of parts over which a set spans. Since the sets that span more than one part bring extra communication and maintenance complexities in parallel computations, especially when the mesh changes, the sets with the single part constraint will be the focus of this paper.

2.3.2 Part boundaries

Each part is treated as a serial mesh with the addition of mesh part boundaries to describe groups of mesh entities that are on the links between parts, where mesh entities on part boundaries, called *part boundary entities*, are duplicated on all parts for which they bound other higher order mesh entities. Mesh entities that are

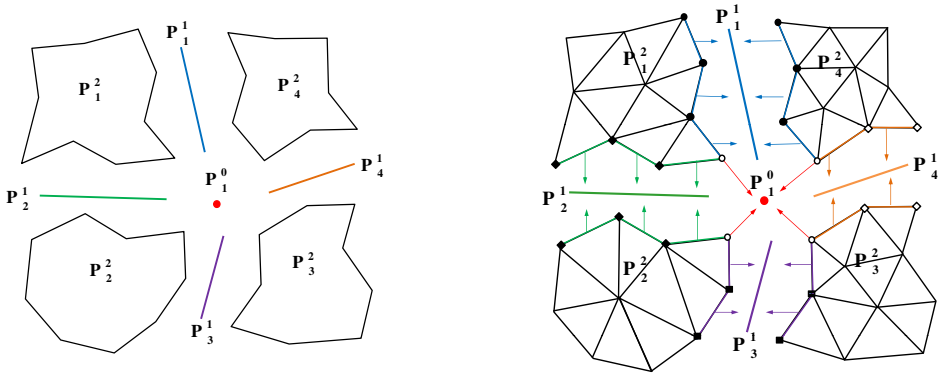

Fig. 2 Distributed mesh and its association with the partition model via partition classifications.

not on any part boundaries exist on a single part and termed as *interior* mesh entities. In the 2D mesh illustrated in Figure 1, the part boundary entities are the vertices and edges that are adjacent to mesh faces on different parts.

For each mesh entity, the *residence part operator* [27, 28] returns a set of part id(s) where a mesh entity exists based on adjacency information: If mesh entity M_i^d is not adjacent to any higher dimension entities, the residence part of M_i^d is the id of the single part where M_i^d exists. Otherwise, the residence part of M_i^d is the set of part id's of the higher order mesh entities that are adjacent to M_i^d. Note that part boundary entities share a set of residence parts depending on the locations in part boundaries.

2.3.3 Partition model

For the purpose of topological representation of a mesh partitioning and efficient parallel operations, a partition model is developed as a conceptual model existing between a geometric model and a mesh. A partition model consists of partition model entities [27,28].

Partition (model) entity: a topological entity in the partition model, P_i^d, which represents a group of mesh entities of dimension d or less, which have the same residence part(s).

Partition classification: the unique association of mesh entities to partition model entities.

Reverse partition classification: for each partition entity, the set of equal order mesh entities classified on that entity.

Figure 2 depicts the 2D distributed mesh in Figure 1 and its association with the partition model. The mesh entity arrows indicate the partition classification of the mesh entities onto the partition entities. For instance, the mesh vertex M_i^0, duplicated on four parts, is classified on the partition vertex P_1^0. Other mesh vertices and edges (like M_j^1) duplicated on part boundaries are classified on partition edges.

2.3.4 Mesh partitioning

Mesh partitioning is a procedure to perform mesh distribution to a number of parts in which the amount of computational work required for each part is equal and the amount of inter-part communications is minimized. The computational work is often associated with certain *objects* in the computation. In the case of unstructured mesh applications, objects can be the mesh entities (vertices, edges, faces, and regions) and decompositions can be computed with respect to any of these entities or to a combination of entities (e.g., vertices and regions) [29,30]. In the current discussion, a *partition object* is defined as follows:

Partition object: the basic unit to which a destination part id can be assigned for mesh partitioning. It can be either a mesh entity that is not on the boundary of any higher dimension mesh entities, or a mesh set contained in a single part.

Graph/hypergraph-based algorithms are effective for unstructured mesh partitioning [30–33].

2.3.5 Mesh migration

Mesh migration is a procedure that moves mesh entities from part to part to support *(i)* mesh distribution to parts, *(ii)* mesh load balancing, or *(iii)* obtaining mesh entities needed for mesh modification operations [28, 34]. An efficient mesh migration algorithm with minimum resources (memory and time) and parallel operations designed to maintain the mesh load balance throughout the computation is an important factor for pursuing high performance in parallel adaptive simulations. To migrate mesh entities to another part, the destination part id must be specified to each partition object before moving the mesh entities.

2.4 Generic Components for Parallel Adaptive Simulations

The discussion on the data model and requirement analysis in Sections 2.1 - 2.3 necessitates the following three generic components to support mesh level operations in adaptive mesh-based simulations on a massively parallel computing environment:

Set: component for grouping arbitrary data with common set requirements.
Iterator: component for iterating over a range of data.
Tag: component for attaching arbitrary user data to arbitrary data or set with common tagging requirements.

A generic component can be reused in various situations in which the concepts of the component are met as a minimal set of requirements and associated types. For instance, the set component can be used on geometric entities, or mesh entities, and so on. The iterator and tag components can be used on the geometric model, the mesh, or sets.

Sections 3 - 5 present the design and implementation of the three components - set, iterator, and tag - which are in use as utilities. They also illustrate how the three generic components were used to implement various iterator, set and tagging needs in the Flexible distributed Mesh DataBase (FMDB) and Geometric Model Interface (GMI), a distributed mesh and geometric model data infrastructure, to support parallel adaptive simulations [13,27,28].

3 Set Component

A set is a collection of objects or a container of data objects where a set can contain other sets. Data objects from which a given set is composed are called *elements* or *members* of the set. Each non-set data member in a set can have relations to each other. A common relation useful in most applications is *ordering*. Depending on the need to preserve the insertion order for non-set data members, two set types are defined [16,18,23]:

Ordered set: if any two data members are comparable in terms of the insertion ordering, a set is an ordered set. An ordered set can contain duplicate data members.
Unordered set: if the insertion ordering is not preserved, a set is an unordered set. An unordered set contains unique data members.

3.1 Design and Implementation

To support the set related requirements of parallel adaptive simulations, the primary constituents of the set component include: *(i) set* for holding multiple arbitrary data of the same type, *(ii) set holder* for maintaining all active sets uniquely identified by the set handles, and *(iii) settable object* which models data containable in a set.

The syntax of set API's for basic set operations, such as set creation/deletion, set existence/type check, set data insertion/deletion, is listed in the following. Two Standard Template Library (STL) containers [1, 35], *stl::list* and *stl::vector*, are used as data types of the output arguments with multiple data. The output of each API function is an integer value of either success (*zero*) or failure (predefined non-zero error code).

```
// create set and store it in set holder
// type: ordered or unordered
template<typename Entity>
int SetHolder_CreateSet (SetHolder<Entity>*,
        int type, Set<Entity*>*);

// delete set and remove it from set holder
template<typename Entity>
int SetHolder_DelSet (SetHolder<Entity>*,
        Set<Entity>*);

// check whether set exists in set holder
template<typename Entity>
int SetHolder_HasSet (SetHolder<Entity>*,
        Set<Entity>*, int*);

// get a list of sets contained in set holder
template<typename Entity>
int SetHolder_GetSet (SetHolder<Entity>*,
        vector<Set<Entity>*>&);

// get # sets contained in set holder
template<typename Entity>
int SetHolder_GetNumSet (SetHolder<Entity>*, int*);

// get type of set (ordered or unordered)
template<typename Entity>
int Set_GetType (Set<Entity>*, int*);

// check whether set has data
template<typename Entity>
int Set_HasEnt (Set<Entity>*, Entity*, int*);

// get # data contained in set
template<typename Entity>
int Set_GetNumEnt (Set<Entity>*, int*);

// insert data into set
template<typename Entity>
int Set_AddEnt (Set<Entity>*, Entity*);

// insert multiple data into set
template<typename Entity>
int Set_AddEntArr (Set<Entity>*, vector<Entity*>&);
```

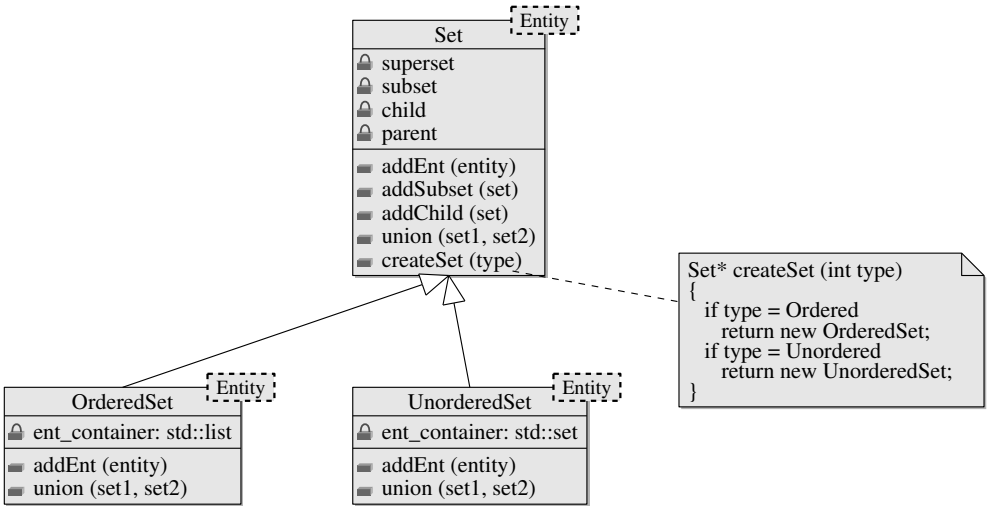

Fig. 3 Class diagram of set component.

```
// remove data from set
template<typename Entity>
int Set_RmvEnt (Set<Entity>*, Entity*);

// remove multiple data from set
template<typename Entity>
int Set_RmvEntArr (Set<Entity>*, vector<Entity*>&);
```

Herein, *Entity* is a concept modeling a piece of data in a data model. In the current unstructured mesh applications, *Entity* can be a mesh entity in a mesh, or a model entity in a geometric model. Given a desired set type (ordered or unordered), the function *SetHolder_CreateSet* creates a set and stores its handle in the set holder object provided.

To implement the data uniqueness and order preservation characteristics of ordered/unordered sets efficiently, *STL::set* and *STL::list* containers are ideal for an unordered set and an ordered set, respectively. The *factory method* design pattern [36], an object-oriented design method to define an interface for creating a class object with yielding instantiation to sub-classes with specialization, is used to support the dynamic container data structure selection upon the set type input at runtime. As illustrated in Figure 3, the base class (*Set*) provides a uniform interface (function *createSet*) and the factory method enables creation of ordered or unordered set dynamically.

In addition to the general set functionality described above, it's also desirable to support set relations (parent/child and superset/subset) and set binary operations such as union, subtraction, and intersection for flexible set manipulation. The following are the API's for set relations.

```
// superset-subset operations
template<typename Entity>
int Set_AddSuperSub (Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_RmvSuperSub (Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_HasSub (Set<Entity>*, Set<Entity>*, int*);

template<typename Entity>
int Set_GetNumSuper (Set<Entity>*, int*);

template<typename Entity>
int Set_GetNumSub (Set<Entity>*, int*);

template<typename Entity>
int Set_GetSuper (Set<Entity>*, list<Set<Entity>*>&);

template<typename Entity>
int Set_GetSub (Set<Entity>*, list<Set<Entity>*>&);

// parent-child operations
template<typename Entity>
int Set_AddPrntChld (Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_RmvPrntChld (Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_IsChldOf (Set<Entity>*, Set<Entity>*, int*);

template<typename Entity>
int Set_GetNumPrnt (Set<Entity>*, int*);

template<typename Entity>
int Set_GetNumChld (Set<Entity>*, int*);

template<typename Entity>
int Set_GetPrnt (Set<Entity>*, list<Set<Entity>*>&);

template<typename Entity>
int Set_GetChld (Set<Entity>*, list<Set<Entity>*>&);
```

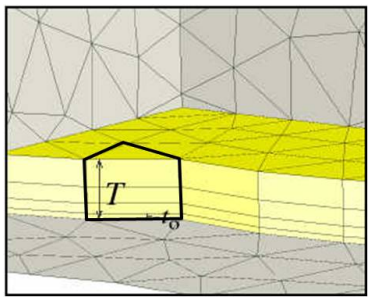

Fig. 4 Part of a boundary layer mesh (the stacks of mesh entities considered as entity sets).

The following are the API's for set binary operations.

```
// set binary operations: third argument is output set
template<typename Entity>
int Set_Unite (Set<Entity>*,
        Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_Intersect (Set<Entity>*,
        Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_Subtract (Set<Entity>*,
        Set<Entity>*, Set<Entity>*);
```

3.2 Application

Due to the property of a mesh entity, which must be assigned to a part for management, there are two types of entity sets used in unstructured mesh applications: *(i)* a set of entities from a single part, and *(ii)* a set of entities from multiple parts.

P-set: a mesh entity set with the single part restriction that all mesh entities in the set belong to the same part.

NP-set: a mesh entity set without the single part restriction.

The user can designate a P-set as a partition object and then migrate all entities contained in the P-set and their adjacent higher dimension entities to the destination part during the migration procedure.

In support of parallel anisotropic adaptive meshing based on mesh metric fields with adaptive boundary layer meshes [37], a stack of boundary mesh entities is required to be on a single part for mesh modification, partition and migration. In Figure 4, mesh entities contained in the black polygon illustrate a stack of entities to be treated with a P-set.

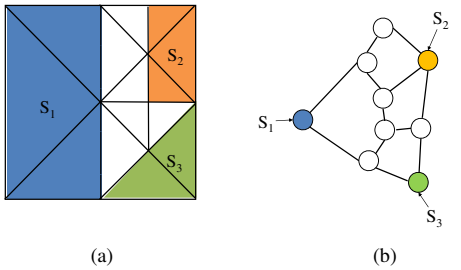

Fig. 5 Distributed mesh with three P-sets and partition object diagram.

3.2.1 Mesh entity set for iMesh:

Using the set component with a mesh instance as a *SetHolder*, all requirements of *iMesh* sets are easily supported [16]; *(i)* populating by addition or removal of entities into or from the set, *(ii)* traversal through an iterator with various conditions such as type, or topology of an entity, *(iii)* set binary operations (union, subtraction, intersection), and *(iv)* relationships among sets such as superset/subset and parent/child.

3.2.2 Mesh entity set for boundary layer adaptation:

The requirements of a P-set for supporting boundary layer mesh adaptations are the following.

1. mesh entities contained in a set are unique.
2. mesh entities are ordered in a set based on their insertion order.
3. mesh entities contained in a set are not part of the boundary of any higher dimension mesh entities.
4. user can attach arbitrary data to a set.
5. iterating over mesh entities in a set.
6. migrating a set and constituting entities to another part.
7. migrating tag data along the set migration.

Items 1-5 are supported by using the set component with a part as a *SetHolder*. For Item 6 and 7, the user can designate a set as a partition object so the mesh entities contained in the set and tag data attached to the set are migrated along when the set is migrated to another part.

Figure 5(a) depicts a 2D mesh with three P-sets, each of which consists of mesh faces and is designated as a partition object. Figure 5(b) illustrates its corresponding partition object diagram, where graph nodes (circles) represent partition objects and graph edges (lines) represent mesh edge based adjacencies between partition objects.

To support boundary layer mesh adaptation in parallel, the mesh migration algorithm developed in Reference [27,28] was improved to migrate both mesh entities and P-sets.

```
Data: EntitiesToMigr[d]
Result: entities and P-sets are migrated to destination
        parts
begin
    /* STEP 1: pack messages and send                    */
    for each M_i^d ∈ EntitiesToMigr[d] do
        pack message A (information of M_i^d, ...);
        if FMDB_Ent_IsInSet(M_i^d) = false then
            isInSetFlag ← false;
            append (isInSetFlag) to message A;
        else
            isInSetFlag ← true;
            S_i ← FMDB_Ent_GetSet(M_i^d);
            entPos ← FMDB_Set_GetEntOrder(S_i, M_i^d);
            append (isInSetFlag, S_i, entPos) to message A;
        end
        P_local sends message A to P_i;
    end
    /* STEP 2: initialize a map to track P-sets          */
    PSetMap ← map<pair< P_i, S_i >, S_j >;
    /* STEP 3: unpack messages and create entities and
    P-sets                                               */
    while P_i receives message A from P_sender do
        M_i^d' ← FMDB_Ent_Create(P_i, entity info);
        if isInSetFlag = false then continue;
        if PSetMap [pair< P_sender, S_i >] exists then
            S_j ← PSetMap [pair< P_sender, S_i >];
            FMDB_Set_AddEnt(S_j, M_i^d', entPos);
        else
            S_i' ← FMDB_Set_Create(P_i, PSET);
            PSetMap [pair< P_sender, S_i >] ← S_i';
            FMDB_Set_AddEnt(S_i', M_i^d', entPos);
        end
    end
end
```

Algorithm 1. The mesh entity exchange procedure to transfer mesh entities and P-sets to destination parts.

Given an array of entities of dimension d to migrate, the pseudo code of Algorithm 1 illustrates the mesh entity exchange procedure to transfer mesh entities and P-sets to destination parts in mesh migration. Note each P-set handle is attached (tagged) to each consisting entity to expedite the P-set manipulation and migration. The FMDB API *FMDB_Ent_IsInSet* is to check whether an entity is contained in a specific set or not. The main steps in Algorithm 1 are listed below:

Step 1: pack messages and send to destination parts. Before sending a message (A) of a mesh entity (M_i^d) in *EntitiesToMigr*[d], check if the entity is in a set (say, S_i), and pack the message A with a variable (*isInSetFlag*) to indicate whether the message contains set data or not. If yes, i.e. *isInSetFlag* equals *true*, pack the set data within the message A also.

Step 2: initialize a set map to keep track of P-sets. On each process, create a set map *PSetMap* to store the pairwise relations between an original set (S_i) from the sending part (P_i) and its local copy (S_j).

Step 3: unpack messages and create entities and P-sets. When a part P_i receives the message A from the sender (P_{sender}), it creates a new mesh entity $M_i^{d'}$. Then it checks the variable *isInSetFlag* contained in the message A. If necessary, i.e. *isInSetFlag* equals *true*, it searches *PSetMap* for a local copy of the set S_i from the part P_{sender}. If a local copy S_j exists, the new entity is added into the existing set S_j. Otherwise, a new set S_i' is created, a new entry for the set S_i' is added into *PSetMap*, and $M_i^{d'}$ is added into the new set S_i'.

As illustrated in Algorithm 1, the FMDB API naming convention [38] is to have the operation target in the middle of the function name and the operation performed on the target data in the end. For instance, *(i)* the function with *Ent* in the middle of the name, such as *FMDB_Ent_IsInSet* and *FMDB_Ent_GetSet*, is performed on a specific mesh entity, *(ii)* the function with *Part* in the middle of the name is performed on a specific part, and *(iii)* the function with *Set* in the middle of the name, such as *FMDB_Set_AddEnt* and *FMDB_Set_Create*, is performed on a specific set.

4 Iterator Component

An iterator is a generalization of a pointer, an object that points to another object used to traverse over a range of objects. When an iterator is within a range, the increment operator moves the iterator to the next object [1,35]. In the Standard Template Library (STL), the concept *iterator* models an object which traverses over a single one-dimensional container [1,35].

4.1 Design and Implementation

In addition to the general iterator functionality such as initialization, advancement, reset, position check, and iterator deletion, the iterator component for parallel adaptive simulations should support: *(i) filtering*: skipping unwanted data over traversal with various conditions specifiable by the user, and *(ii) resilience*: validity with data modification such as data insertion or deletion.

The following are the API's for iterator initialization, advancement, reset, the position check, and iterator deletion.

```
// iterator initialization
template<typename Iterator, typename Entity>
int Iter_Init (const Iterator& first,
    const Iterator& last,
    int type, int topo, void* ptr,
    void (*functor)(Iterator&,Iterator&,void*,int,int),
    Iter<Iterator, Entity>*);

// iterator advancement
template<typename Iterator, typename Entity>
int Iter_GetNext (Iter<Iterator, Entity>*, Entity*);

// iterator reset
template<typename Iterator, typename Entity>
int Iter_Reset (Iter<Iterator, Entity>*);

// iterator position check
template<typename Iterator, typename Entity>
int Iter_IsEnd (Iter<Iterator, Entity>*, int*);

// iterator deletion
template<typename Iterator, typename Entity>
int Iter_Del (Iter<Iterator, Entity>*);
```

Herein, the concept *Entity* represents a piece of data in a data model. For instance, it can be a mesh entity in a mesh, a model entity in a geometric model, or a mesh entity in a set. In a distributed mesh environment, an iterator traverses mesh entities within a single local part to avoid extra communication costs.

In *Iter_Init* function, given the input arguments consisting of *(i)* a data range represented by an iterator pair [*first*, *last*)[2], *(ii)* type (dimension), *(iii)* topology, *(iv)* *void** type pointer (*ptr*), and *(v)* user-defined filtering function pointer (*functor*), an iterator instance is created and returned. The iterator traverses the entity data within the data range that satisfies the specified type, topology and filtering function pointer requirements. The *void** type pointer *ptr* is reserved for any user-defined data structure. If unnecessary, it is set to *null*.

The function *Iter_GetNext* fetches the entity data pointed by the current iterator, and then advances the iterator to the next available entity data. If the iterator reaches to the end of the data range, the function *Iter_GetNext* returns non-zero error code.

Unlike an STL iterator [35], the iterator component for adaptive unstructured mesh simulations should be able to traverse multiple containers through a single iterator since the topological model data is usually stored in multiple containers per type. For instance, mesh and model entities can be stored in four containers per type: vertices, edges, faces and regions. To support traversing multiple containers with a single iterator, a *linking* method is developed in the iterator advancement operator with which the end of the previous one-dimensional container is connected to the beginning of the next container.

The following is the pseudo-code to traverse multiple containers with a single iterator.

```
// cur_pos is the current iterator position
if cur_pos == current_container.end
    if (current_container.end != data_range.end)
        cur_pos = next_container.begin;
    else
        return;
else
    advance cur_pos;
```

4.2 Application

To support parallel adaptive simulations, the minimum set of iterators needed in a mesh and geometric model includes *(i)* mesh entity traversal in a part by type and/or topology, *(ii)* mesh entity traversal by reverse geometric classification, *(iii)* part boundary mesh entity traversal, *(iv)* mesh entity traversal in a set by type and/or topology, and emph(v) geometric entity traversal in the entire geometry by type. For all cases, the iterator range for mesh entities is limited to entities on a single part or a set.

In the FMDB, a single part uses an *stl::set* container to store unique mesh entities. As per *stl::set* iterator specification [35], an iterator is guaranteed to work properly even with mesh modifications except for the case when the entity being pointed by the iterator is deleted. However, when a mesh entity currently being pointed by the iterator is deleted, the invalid iterator problem can be avoided by advancing the iterator on deleted entity to the next.

4.2.1 Mesh entity traversal in part by dimension and/or topology:

An iterator for a mesh entity traversal is implemented by providing the first and one past the end of an entity container as for the data range based on the requested entity dimension and topology. To traverse mesh entities in a part by type and/or topology, various combinations of the input type and topology pairs are possible for entity filtering. Users can also specify all types (*ALLTYPE*) and all topologies (*ALLTOPO*).

[2] The notation [*first*, *last*) refers to all the iterators from *first* up to, but not including, last [35].

The following FMDB API illustrates how to initialize an iterator to implement a mesh entity traversal in a part. For each part, the entity container of dimension i is denoted as *part→container*[i] where i={VERTEX, EDGE, FACE, REGION}. For each entity container of dimension i, *part→container*[i].*begin* and *part→container*[i].*end* denote the first and one past the end of the container, respectively. *pPart* is a pointer type to a part, and *mEntity* is a class type of mesh entities.

```
typedef entity_container_iterator_type part_iter;
typedef Iterator<part_iter, mEntity>* pPartEntIter;

// iterator initialization
int FMDB_PartEntIter_Init (pPart part, int type,
        int topo, pPartEntIter& iter)
{
  if type==ALLTYPE
    return Iter_Init (part->container[VERTEX].begin,
          part->container[REGION].end, type, topo,
          (void*)part, &EntityProcessFilter, iter);
  else
    return Iter_Init (part->container[type].begin,
          part->container[type].end, type, topo,
          (void*)part, &EntityProcessFilter, iter);
}
```

Herein, if the input entity type is *ALLTYPE*, the iterator input range is specified as the mesh entities of all dimensions, denoted by *part→container*[*VERTEX*].*begin* and *part→container*[*REGION*].*end*. Otherwise, the iterator input range is specified as mesh entities of a specific dimension, denoted by *part→container*[*type*].*begin* and *part→container*[*type*].*end*.

For readability, the predefined data type of an entity container iterator on each part is *part_iter* and *pPartEntIter* is a pointer to the template class *Iterator<part_iter, mEntity>*. The following pseudo code illustrates the function *EntityProcessFilter*, which moves the iterator to the next proper position that is a pointer to the next mesh entity satisfying the type and topology criterion.

```
void EntityProcessFilter (part_iter& ibegin,
      part_iter& iend, void* ptr, int type, int topo)
{
  if ibegin==iend
    return;
  for each entity in range [ibegin, iend)
    if entity->type==type && entity->topology==topo
      ibegin = current_entity_position;
      return;
  ibegin=iend;
}
```

The following code illustrates how the iterator component is used to implement iterator advancement, position check, deletion and reset functionalities in a mesh entity traversal in a part.

```
// iterator advancement
int FMDB_PartEntIter_GetNext
     (pPartEntIter iter, mEntity* ent)
{
  return Iter_GetNext (iter, ent);
}

// iterator position check
int FMDB_PartEntIter_IsEnd
     (pPartEntIter iter, int* isEnd)
{
  return Iter_IsEnd(iter, isEnd);
}

// iterator deletion
int FMDB_PartEntIter_Del (pPartEntIter iter)
{
  return Iter_Del (iter);
}

// iterator reset
int FMDB_PartEntIter_Reset (pPartEntIter iter)
{
  return Iter_Reset (iter);
}
```

4.2.2 Mesh entity traversal by reverse classification:

For the input geometric entity of dimension d, mesh entity traversal by reverse geometric classification is implemented using the iterator component with the filtering function that checks the geometric classification of each mesh entity on traversal. Note the cost of the reverse classification through iterator is *O(n)* where n is the number of mesh entities in a part, while the reverse classification can be obtained in *O(1)* cost using mesh adjacencies.

The following FMDB API and the implementation illustrate how to initialize an iterator for a mesh entity traversal by reverse classification.

```
// iterator initialization
int FMDB_PartEntIter_InitRevClas (pPart part,
        pGeomEnt geomEnt, int type, pPartEntIter& iter)
{
  if type==ALLTYPE
    return Iter_Init (part->container[VERTEX].begin,
        part->container[REGION].end, type, topo,
        (void*)part_ent, &GeomClasProcessFilter, iter);
  else
    return Iter_Init (part->container[type].begin,
        part->container[type].end, type, topo,
        (void*)part_ent, &GeomClasProcessFilter, iter);
}
```

Herein, *pGeomEnt* is a pointer type to a geometric entity. If the input entity type is *ALLTYPE*, the iterator input range is specified as the mesh entities of all dimensions. Otherwise, the iterator input range is specified as mesh entities of a specific dimension. Using a data structure *part_ent* that is casted into *void∗* type and contains a pair of local part and specific geometric entity, the input filtering function *GeomClasProcessFilter* is used to increment the iterator to the next proper position which points to a mesh entity classified on a given geometric entity *geomEnt*. Note that a local part stored in the data structure *part_ent* is used by the linking method to traverse multiple entity containers in the part. The following is the pseudo code of function *GeomClasProcessFilter*.

```
void GeomClasProcessFilter(part_iter& ibegin,
      part_iter& iend, void* ptr, int type, int topo)
{
  ent = (cast<part_geomEnt*> ptr)->second;
  if ibegin==iend
    return;
  for each entity in range [ibegin, iend)
    if entity->geometric_classification
                    ==cast<pGeomEnt> ent
      ibegin = current_entity_position;
      return;
  ibegin=iend;
}
```

An example usage is to calculate the number of mesh entities classified on a specific geometric entity through the FMDB API *FMDB_GeomEnt_GetNumRevClas*. The pseudo code of Algorithm 2 illustrates the procedure, which consists of three main steps: *(i)* initializing an iterator based on the input part, geometric entity (*geomEnt*) and entity type, *(ii)* advancing the iterator one step forward in the entity traversal loop, and *(iii)* deleting the iterator to avoid memory leak.

4.2.3 Part boundary mesh entity traversal:

A part boundary entity traversal is implemented using the iterator component with the filtering function which checks the duplicated copy existence of each mesh entity (also referred to as *remote copy* [27,28]) on traversal. Given the input arguments consisting of entity type, topology and non-local part id, called *target_part_id*, the iterator creation API initializes an iterator to traverse mesh entities duplicated on the part boundary between the local part and target part.

```
int FMDB_PartEntIter_InitPartBdry (pPart part,
     int target_part_id, int type, int topo,
     pPartEntIter& iter)
```

Data: part, geomEnt, type
Result: store the number of mesh entities classified on geomEnt into numEnt
begin
    /* STEP 1: initialize an iterator */
    numEnt ← 0;
    iterEnd ← FMDB_PartEntIter_InitRevClas(part, geomEnt, type, iter);
    /* STEP 2: traverse mesh entities in a loop */
    while iterEnd = *false* **do**
        iterEnd ← FMDB_PartEntIter_GetNext(iter, meshEnt);
        if iterEnd = *true* **then** break;
        ++numEnt;
    end
    /* STEP 3: delete the iterator */
    FMDB_PartEntIter_Del(iter);
    return numEnt;
end

Algorithm 2. Example of mesh entity traversal by reverse classification: to calculate the number of mesh entities classified on a specific geometric entity in a part.

```
{
  part_pid=pair<part, target_part_id>;
  if type==ALLTYPE
    return Iter_Init (part->container[VERTEX].begin,
        part->container[REGION].end, type, topo,
        (void*)part_pid, &PartBdryProcessFilter, iter);
  else
    return Iter_Init (part->container[type].begin,
        part->container[type].end, type, topo,
        (void*)part_pid, &PartBdryProcessFilter, iter);
}
```

Using a data structure *part_pid* that is casted into *void∗* type and contains a pair of local part and target part id, the function *PartBdryProcessFilter* moves the iterator to the next proper position, which points to a mesh entity that is on the part boundary between the local part and target part with the help of partition classification.

4.2.4 Mesh entity traversal in set by type and/or topology:

Similar to a mesh entity iterator in a part described in §4.2.1, a mesh entity iterator in an entity set is implemented using the iterator component with the input range [*set→container.begin*, *set→container.end*), and the function *EntityProcessFilter*, where *set→container* denotes the entity container for a set.

4.2.5 Geometric entity traversal in the entire geometry by type:

In the Geometric Model Interface (GMI) [39], an *stl::vector* container is used to store all the model entities of a given dimension. Given the geometric model (*model*) and entity dimension (*type*), the following GMI API illustrates how to initialize an iterator for a model entity traversal.

```
typedef model_entity_container_iterator_type model_iter;
typedef Iterator<model_iter, GEntity>* pGMdlEntIter;

// iterator initialization
int GMI_MdlEntIter_Init (pGModel model, int type,
          pGMdlEntIter& iter)
{
  if type==ALLTYPE
    return Iter_Init (model->container[VERTEX].begin,
        model->container[REGION].end, type, 0,
        (void*)model, &GEntityProcessFilter, iter);
  else
    return Iter_Init (model->container[type].begin,
        model->container[type].end, type, 0,
        (void*)model, &GEntityProcessFilter, iter);
}
```

Herein, *pGModel* is a pointer type to a geometric model, and *GEntity* is a class type to geometric entities. For readability, the predefined data type of an iterator on a model entity container is *model_iter* and *pGMdlEntIter* is a pointer to the template class *Iterator<model_iter, GEntity>*.

Similar to a mesh entity iterator in a part described in §4.2.1, the model entity container of dimension i is denoted as *model→container*[i] where i={VERTEX, EDGE, FACE, REGION}. If the input entity type is *ALLTYPE*, the iterator input range is specified as the model entities of all dimensions, from $VERTEX$ to $REGION$ type. Otherwise, the iterator input range is specified as the model entities of a specific dimension. The input entity filtering function *GEntityProcessFilter* moves the iterator to the next proper position in the geometric model.

The following code illustrates how the iterator component is used to implement iterator advancement, reset, the position check, and deletion functionalities for a model entity traversal in the entire geometric model.

```
// iterator advancement
int GMI_MdlEntIter_GetNext
     (pGMdlEntIter iter, GEntity* ent)
{
  return Iter_GetNext (iter, ent);
}

// iterator position check
int GMI_MdlEntIter_IsEnd
     (pGMdlEntIter iter, int* isEnd)
{
  return Iter_IsEnd(iter, isEnd);
}

// iterator deletion
int GMI_MdlEntIter_Del (pGMdlEntIter iter)
{
  return Iter_Del (iter);
}

// iterator reset
int GMI_MdlEntIter_Reset (pGMdlEntIter iter)
{
  return Iter_Reset (iter);
}
```

5 Tag Component

Tags are used as containers of arbitrary user-defined data attachable to the geometric model, geometric entities, mesh instance, part, mesh entities, sets, and fields. Different values of a particular tag can be associated with different data models, entities or sets [16,18,23].

5.1 Design and Implementation

The tag component consists of *(i) tag data* for representing arbitrary user data, *(ii) tag handle* for holding a unique tag identifier attachable to data, *(iii) tag holder* for maintaining all active tags identifiable with handles, and *(iv) taggable object* which models the data to which tag data is attached with a tag handle.

Each tag handle is uniquely identified by a pair of belonging tag holder and string tag name. A tag handle has two attributes, *(i) tag data type* which is primary type (integer, double, entity and set) or arbitrary type data, and *(ii) tag size* which specifies the number of data in tag data. If the tag size is 1, the tag data holds one single piece of data of a given tag type. If the tag size is greater than 1, the tag data holds an array of data of a given tag type.

The following are the API's for a tag holder, including tag handle creation/deletion and tag handle queries in a tag holder.

```
// given tag name,  type, and  size,
// create tag handle and store it in tag holder
TagHandle* TagHolder_CreateTag
    (TagHolder*, const char* name, int type, int size);
```

```
// delete tag handle and remove it from tag holder
int TagHolder_DelTag (TagHolder*, TagHandle*);

// remove all tags from tag holder
void TagHolder_ClearTag (TagHolder*);

// check tag type matches given type info
int TagHolder_CheckTag (TagHolder*, TagHandle*, int type);

// check tag exists in tag holder
int TagHolder_HasTag (TagHolder*, TagHandle*, int*);

// given tag name, find tag from tag holder
int TagHolder_FindTag (TagHolder*,
        const char* name, TagHandle*);

// get a list of tags stored in tag holder
void TagHolder_GetTag (TagHolder*, vector<TagHandle*>&);
```

For a taggable object and tag handle, the API's for getting/setting a single or array of tag data with a primary type (integer, double, entity and set) are the following.

```
template<typename Type>
int Taggable_SetData (Taggable*, TagHandle*, Type*);

template<typename Type>
int Taggable_GetData (Taggable*, TagHandle*, Type[]);
```

For a taggable object and tag handle, the API's to get/set a single or array of *void∗* tag data are presented below. Note that template typename *Type* enables transforming *void∗* type tag data to specific primary type data specified on the tag handle creation.

```
template<typename Type>
void Taggable_SetByteData (Taggable*, TagHandle*,
        const void* data, int size);

template<typename Type>
void Taggable_SetByteArrData (Taggable*, TagHandle*,
        const void* data, int size);

template<typename Type>
void Taggable_GetByteData (Taggable*, TagHandle*,
        void** data, int* size);

template<typename Type>
void Taggable_GetByteArrData (Taggable*, TagHandle*,
        void** data, int* size);
```

For a taggable object and tag handle, the API to delete tag data is presented below.

```
void Taggable_DelTag (Taggable*, TagHandle*);
```

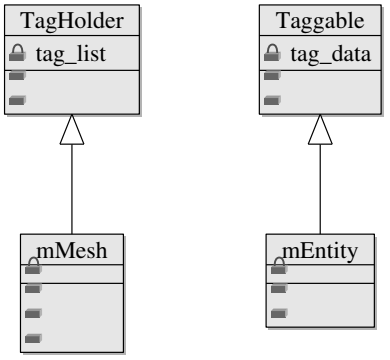

Fig. 6 Class diagram of the mesh and mesh entity.

5.2 Application

To support tag handles created per mesh instance, the mesh instance class, *mMesh*, inherits from the class *TagHolder*. To support efficient tag data access and automatic tag data removal along the taggable object deletion, classes of part, mesh set and mesh entity inherit from the class *Taggable*. The class diagram in Figure 6 illustrates *mMesh* (the mesh instance class) inherited from *TagHolder* and *mEntity* (the mesh entity class) inherited from *Taggable*, which are to support tag handles created in a mesh instance and tag data attachable to mesh entities, respectively.

The following is the pseudo code to create a tag handle of a specific name, type and size. *pMeshMdl* is a pointer type to a mesh instance.

```
int FMDB_Mesh_CreateTag (pMeshMdl mesh,
        const char* name, int type,
        int size, TagHandle* tag)
{
  if tag with given name exists in mesh
    tag = existing_tag;
  else
    tag = TagHolder_CreateTag(cast<TagHolder*>(mesh),
                        name, type, size);
  return SUCCESS;
}
```

If a tag handle of the specific name, type and size already exists in the tag holder object, i.e. the mesh, the function returns the existing tag handle. Otherwise, it creates a new tag handle and stores it in the mesh.

The following is the pseudo code to delete a specific tag handle from the mesh.

```
int FMDB_Mesh_DelTag (pMeshMdl mesh, TagHandle* tag)
{
  TagHolder_DelTag(cast<TagHolder*>(mesh), tag);
  delete tag;
  return SUCCESS;
}
```

The following is the pseudo code to search and get tag handle(s).

```
// for input string name, find tag handle from mesh
int FMDB_Mesh_FindTag (pMeshMdl mesh, char* name,
      TagHandle* tag)
{
  return TagHolder_FindTag (cast<TagHolder*>(mesh),
      name, tag);
}

// get all tag handles in mesh
int FMDB_Mesh_GetTag (pMeshMdl mesh,
      vector<TagHandle*>& tags)
{
  return TagHolder_GetTag (
      cast<TagHolder*>(mesh), tags);
}
```

The FMDB APIs to set/get integral tag data to a part, and delete tag data attached to a part are illustrated below.

```
// set integral tag data to part
int FMDB_Part_SetIntTag (pPart part, TagHandle* tag,
     const int data)
{
  return Taggable_SetData<int>(cast<Taggable*>(part),
     tag, &data);
}

// get integral tag data from part
int FMDB_Part_GetIntTag (pPart part, TagHandle* tag,
     int* data)
{
  return Taggable_GetData<int>(cast<Taggable*>(part),
     tag, data);
}

// delete tag data attached to part
int FMDB_Part_DelTag (pPart part, TagHandle* tag)
{
  return Taggable_DelTag (cast<Taggable*>(part), tag);
}
```

The code to set/get an array of integral tag data with an entity set is given below. *pEntSet* is a pointer type to a mesh entity set.

```
// get integral tag array data to set
int FMDB_Set_SetIntArrTag (pEntSet set, TagHandle* tag,
        const int* data, int size)
{
  if size!=tag->size return ERROR;
  return Taggable_SetData<int>(cast<Taggable*>(set),
                               tag, data);
}

// get integral tag array data from set
int FMDB_Set_GetIntArrTag (pEntSet set, TagHandle* tag,
        int** data, int* size)
{
  *size = tag->size;
  return Taggable_GetData<int>(cast<Taggable*>(set),
                               tag, *data);
}
```

Contrary to the code to set/get primary type tag data in which accessing the tag data is done in one step, the implementation to set/get *byte* type (*void∗*) tag data is composed of two steps: *(i)* for a given tag handle, retrieve tag type and transform the *void∗* tag data to specific (primary) type data if necessary and *(ii)* if primary type, call single or array type set/get tag functions based on tag size information. For instance, part of the code to set *void∗* tag data with a mesh entity (*ent*) is given below.

```
if tag->type==byte
  out = Taggable_SetByteArrData<char>
      (cast<Taggable*>(ent), tag, data, size);

if tag->type==integer
  if tag->size==1  // single integer
    out = Taggable_SetByteData<int>
      (cast<Taggable*>(ent), tag, data, size);
  else             // multiple integers
    out = Taggable_SetByteArrData<int>
      (cast<Taggable*>(ent), tag, data, size);
```

Herein, if replacing *integer* to other primary data type, the code above can set *void∗* tag data with other primary type, such as *double*, *pEntSet*, or *pMeshEnt* (mesh entity pointer type). As illustrated in the code above, tagging for a part, entity and entity set can be implemented easily by reusing the tag component through the class inheritance and template mechanism.

In parallel computations using the FMDB, migrating the tag data attached to a P-set or an entity along with the migration is achieved through specifying callback functions [27,40] in mesh migration, which is the mechanism to allow the user to specify how to pack the tag data within the entity message packing procedure and how to unpack and attach the tag data received from remote parts within the entity message unpacking procedure. If no callback function is specified, tag data is ignored during the migration and removed automatically when a P-set or an entity is eliminated from a part.

In the migration procedure, callback functions dealing with tag data are defined as pure virtual member functions in a derived class of the FMDB class *pmMigrationCallbacks*, and are called in the entity message packing and unpacking steps. These callback functions *(i) getUserData* specifies the tag data to migrate with a

P-set or an entity, *(ii) receiveUserData* specifies the operations to be performed when the tag data is received on a remote part, and *(iii) deleteUserData* defines how to delete the tag data to avoid memory leak. Since retrieving all tag handles attached to a taggable object is supported, an on-going development effort includes automatic tag data migration along P-set or entity migration.

6 Mesh Adaptive Applications

This section presents two examples in parallel mesh adaptation developed using the geometric model and mesh (Geometric Model Interface (GMI) and FMDB) with the three generic components used underneath as utilities.

6.1 Support Boundary Layer Mesh Adaptation

The boundary layer mesh adaptation procedure [41] is performed on a manifold heat exchanger model, in which a large flow rate comes in a larger tube and dumps into a thin rectangular geometry where the flow is distributed into smaller pipes. The solution based anisotropic adaptation is carried out on this model to capture the flow features.

The initial boundary layer mesh has about $450,000$ regions, and the resulting adapted boundary layer mesh has about 4.5 million regions. The clip-plane views of the interior boundary layer mesh structures before and after the boundary layer mesh adaptation are shown in Figure 7. In the implementation, each stack of boundary layer mesh entities (see the mesh entities in the black polygon in Figure 4) is a P-set, thus all the mesh entities in the set can be handled as a unit and kept together on one part as the mesh changes. These P-sets are then handled as partition objects for mesh partitioning and migration.

6.2 Scaling Studies of Mesh Adaptation

The mesh adaptation procedure developed in SCOREC [34,42] is carried out on a flow simulation example to compare the performance of mesh adaptation between traditional object-oriented and generic programming paradigm. The mesh adaptation procedure is chosen for testing, since it relies heavily on entity iteration and tag data association. The iterator component is to support traversing over either geometric entities or mesh entities, and the tag component is to support data association with either of these entities. Associated data can be arbitrary, such as solution-based mesh size fields.

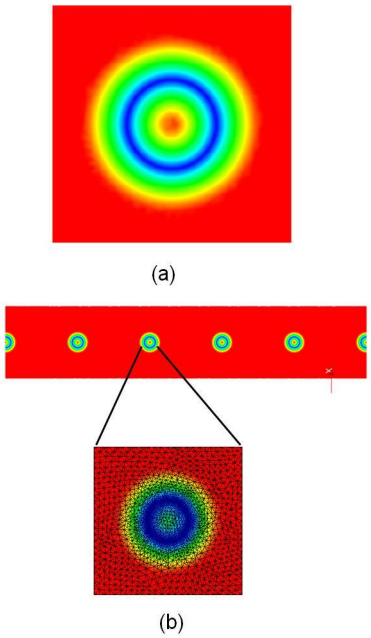

Fig. 8 Defined mesh size field (a) and a segment of straight pipe model with air bubbles (b).

In the adaptive flow simulation, a straight pipe model with air bubbles distributed in the pipe is used (see Figure 8). Figure 8a shows the mesh size field which represents the motion of air bubbles in the geometric model. The smaller size field is shown in blue, implying a fine mesh (or high resolution), and the relative large size field is shown in red, implying a coarse mesh. In multiphase flow simulations, fine meshes at phase boundaries are expected to capture the complicated physical phenomena at the interface [43]. Figure 8b shows a segment of the straight pipe model which involves the motion of five air bubbles by a distance of 1/5 of their radius. A zoomed bubble in the mesh is colored by the magnitude of size field in Figure 8b.

The mesh adaptation procedure starts with an initial uniform tetrahedral mesh with 165 million tetrahedra, and obtains an adapted mesh of 188 million tetrahedra. All the test cases were run on the ANL *Intrepid* (IBM BG/P system) [44]. The test cases were executed on $1,024$ up to $32,768$ cores using 512MB per core memory. The execution time of the mesh adaptation procedure for all the test cases are collected, and scaling factors are computed based on the execution time on $1,024$ cores. The scaling factor is defined as

$$s\text{-}factor \;=\; (np_{base} * t_{base})/(np_i * t_i), \tag{1}$$

where t represents the execution time and np represents the number of cores. For instance, np_{base} represents the

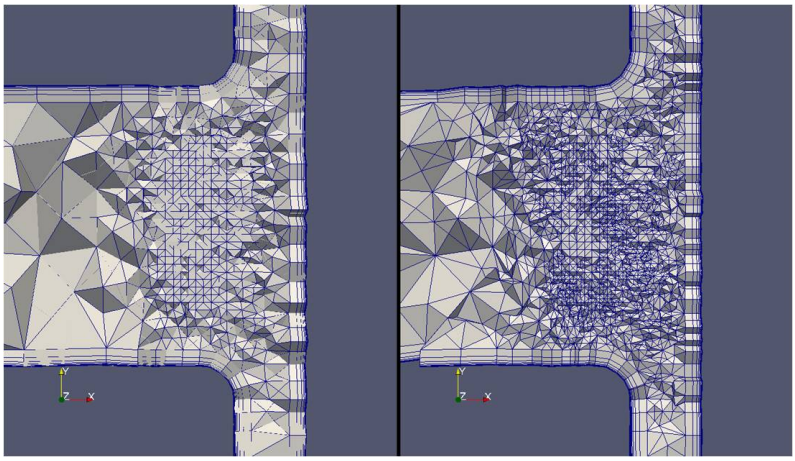

Fig. 7 Clip-plane mesh views of the boundary layer meshes.

Table 1 Scaling results of air-bubble mesh adaptation on ANL Intrepid using two methods: mesh adaptation without using the generic components (*adapt without generic*), and mesh adaptation using the generic components (*adapt with generic*).

	adapt without generic		*adapt with generic*		time
num of cores	time	s-factor	time	s-factor	increase(%)
1,024 (base)	475.90	1	476.07	1	0.04
2,048	329.94	0.72	330.81	0.72	0.26
4,096	220.17	0.54	220.99	0.54	0.37
8,192	107.78	0.55	110.32	0.54	2.36
16,384	74.46	0.40	74.72	0.40	0.35
32,768	44.15	0.34	44.33	0.34	0.41

number of cores in the base case (here is the test case running on $1,024$ cores), and t_{base} represents the execution time of the base case.

The performance results are summarized in Table 1, which compares the performance of two methods including *(i)* the mesh adaptation procedure using the generic components, and *(ii)* the mesh adaptation procedure using the traditional object-oriented programming paradigm. As shown in the last column of the table, the mesh adaptation procedure using the generic components requires at most 2.36% more time than the one without the generic components, and does not affect the scaling. In summary, the generic components achieve code reusability and flexibility without sacrificing the performance of mesh adaptation, compared to the traditional object-oriented programming.

7 Closing Remark

This paper presented the generic iterator, set and tag components developed as part of utilities of data management software tools in unstructured mesh based adaptive simulations, and presented how they were applied to meet various needs of distributed mesh management tools.

In the future, the iterator, set and tagging functionalities will support geometric model and field libraries through well-defined API's. At the same time, for better supporting unstructured mesh applications, more generic components will be designed and developed, such as the *(i)* the *relation* component for relating arbitrary data in different data models when no direct interactions through API's are available, and *(ii)* the *communicator* component for supporting efficient parallel functionalities such as architecture-aware communication and data distribution on massively parallel computers.

On the other hand, more software engineering techniques and generic programming methods can be applied in the software component design for adaptive unstructured mesh simulations. For example, instead of using raw function pointers, the generic iterator component discussed in Section 4 can use *Boost.Function* library [45], thus to allow user greater flexibility in the implementation. However, running the third party software libraries on supercomputers needs to consider the portability issue.

The three generic components developed herein are open source and available at
http://www.scorec.rpi.edu/software.php.

For source code and more information about FMDB and its *iMesh/iMeshP* interfaces, please visit *http://www.scorec.rpi.edu/FMDB*.

Acknowledgements We gratefully acknowledge the support of this work by the Department of Energy (DOE) office of Science's Scientific Discovery through Advanced Computing (SciDAC) institute as part of the Interoperable Technologies for Advanced Petascale Simulations (ITAPS) program, under grant DE-FC02-06ER25769.

References

1. M.H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison Wesley Longman, Reading, MA, 1999.
2. A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional; 1 edition, Feb, 2001.
3. D.R. Musser, G.J. Derge, and A. Saini. *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*. Boston, MA: Addison-Wesley, 2001.
4. B. Stroustrup. *The C++ Programming Language (3rd edition)*. Addison Wesley Longman, Reading, MA, 1997.
5. J.G. Siek and A. Lumsdaine. The Matrix Template Library: generic components for high-performance scientific computing. *Computing in Science & Engineering*, 1(6):70–71, 1999.
6. L.-Q. Lee and A. Lumsdaine. The Generic Message Passing framework. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 10 pp., april 2003.
7. G. Berti. A generic toolbox for the grid craftsman. In *Proceedings of the 17th GAMM Seminar on Construction of Grid Generation Algorithms, Online proceedings athttp://www.mis.mpg.de/conferences/gamm/2001*, 2001.
8. G. Berti. GrAL-the grid algorithms library. *Future Generation Computer Systems*, 22(1-2):110–122, 2006.
9. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. Pract. Exper.*, 30(11):1167–1202, September 2000.
10. P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework. *Computing*, 82(2):103–119, July 2008.
11. P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE. *Computing*, 82(2):121–138, July 2008.
12. F. Cirak and J.C. Cummings. Generic programming techniques for parallelizing and extending procedural finite element programs. *Eng. with Comput.*, 24(1):1–16, January 2008.
13. M.W. Beall and M.S. Shephard. An object-oriented framework for reliable numerical simulations. *Engineering with Computers*, 15:61–72, 1999.
14. M.S. Shephard, M.W. Beall, R.M. O'Bara, and B.E. Webster. Toward simulation-based design. *Finite Elements in Analysis and Design*, 40(12):1575–1598, July 2004.
15. Simmetrix Inc. The Simulation Modeling Suite. Retrieved on 18 Oct. 2011, from http://www.simmetrix.com/.
16. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC). Interoperable Technologies for Advanced Petascale Simulations (ITAPS). Retrieved on 18 Oct. 2011, from http://www.itaps.org/.
17. K.K. Chand, L.F. Diachin, X. Li, C. Ollivier-Gooch, E.S. Seol, M.S. Shephard, T. Tautges, and H. Trease. Toward interoperable mesh, geometry and field components for PDE simulation development. *Engineering with Computers*, 24(2):165–182, November 2007.
18. C. Ollivier-Gooch, L. Diachin, M.S. Shephard, T. Tautges, J. Kraftcheck, V. Leung, X. Luo, and M. Miller. An interoperable, data-structure-neutral component for mesh query and manipulation. *ACM Trans. Math. Softw.*, 37(3):29:1–29:28, September 2010.
19. K.J. Weiler. The radial-edge structure: a topological representation for non-manifold geometric boundary representations. *Geometric Modeling for CAD Applications*, pages 3–36, 1988.
20. M.S. Shephard and M.K. Georges. Reliability of automatic 3D mesh generation. *Computer Methods in Applied Mechanics and Engineering*, 101(1-3):443–462, December 1992.
21. X. Li, M.S. Shephard, and M.W. Beall. Accounting for curved domains in mesh adaptation. *International Journal for Numerical Methods in Engineering*, 58(2):247–276, 2003.
22. M.W. Beall, J. Walsh, and M.S. Shephard. Accessing CAD geometry for mesh generation. In *12th International Meshing Roundtable, Sandia National Laboratories*, pages 2003–3030, 2003.
23. T.J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst. MOAB: A MESH-ORIENTED DATABASE (SANDIA REPORT). Technical report, 2004.
24. Interoperable Technologies for Advanced Petascale Simulations (ITAPS). The ITAPS iMesh Interface Documentation. Retrieved on 18 Oct. 2011, from http://www.itaps.org/software/iMesh_html/index.html.
25. M.W. Beall and M.S. Shephard. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering*, 40(9):1573–1596, 1997.
26. J.-F. Remacle and M.S. Shephard. An Algorithm Oriented Mesh Database. *International Journal for Numerical Methods in Engineering*, 58(2):349–374, sep 2003.
27. E.S. Seol. *FMDB: Flexible distributed Mesh DataBase for parallel automated adaptive analysis*. PhD thesis, Rensselaer Polytechnic Institute, 2005.
28. E.S. Seol and M.S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 22(3-4):197–213, November 2006.
29. J.D. Teresco, K.D. Devine, and J.E. Flaherty. Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In Are Magnus Bruaset and Aslak Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 55–88. Springer Berlin Heidelberg, 2006.
30. M. Zhou, O. Sahni, K.D. Devine, M.S. Shephard, and K.E. Jansen. Controlling Unstructured Mesh Partitions for Massively Parallel Simulations . *SIAM J. Sci. Comput.*, 32:3201–3227, 2010.
31. H.D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2-3):135–148, 1991.

32. K.D. Devine, E.G. Boman, R.T. Heaphy, B.A. Hendrickson, J.D. Teresco, J. Faik, J.E. Flaherty, and L.G. Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, February 2005.
33. K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society.
34. F. Alauzet, X. Li, E.S. Seol, and M.S. Shephard. Parallel anisotropic 3D mesh adaptation by mesh modification. *Engineering with Computers*, 21(3):247–258, January 2006.
35. Silicon Graphics International (SGI. Standard Template Library Programmer's Guide. Retrieved on 22 June 2011, from http://www.sgi.com/tech/stl.
36. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AddisonWesley Professional: 1 edition, Nov, 1994.
37. O. Sahni, K.E. Jansen, M.S. Shephard, C.A. Taylor, and M.W. Beall. Adaptive boundary layer meshing for viscous flow simulations. *Engineering with Computers*, 24(3):267–285, 2008.
38. Scientific Computation Research Center (SCOREC. Flexible distributed Mesh DataBase User's Guide. Retrieved on 18 Oct. 2011, from http://scorec.rpi.edu/FMDB/documentation.html.
39. Scientific Computation Research Center. Simulation Model and Data Management Components. Retrieved on 18 Oct. 2011, from https://www.scorec.rpi.edu/software.php.
40. J.-F. Remacle, O. Klaas, J.E. Flaherty, and M.S. Shephard. Parallel Algorithm Oriented Mesh Database. *Engineering with Computers*, 18(3):274–284, October 2002.
41. A. Ovcharenko, K. Chitale, O. Sahni, K.E. Jansen, and M.S. Shephard. Parallel anisotropic mesh adaptation with boundary layers. *International Journal for Numerical Methods in Engineering (submitted)*, 2012.
42. X. Li, M.S. Shephard, and M.W. Beall. 3D anisotropic mesh adaptation by mesh modification. *Computer Methods in Applied Mechanics and Engineering*, 194(48-49):4915–4950, November 2005.
43. M. Zhou, T. Xie, S. Seol, M.S. Shephard, O. Sahni, and K.E. Jansen. Tools to support mesh adaptation on massively parallel computers. *Engineering with Computers*, April 2011.
44. Argonne National Lab. Quick reference guide. Retrieved on 22 June 2011, from https://wiki.alcf.anl.gov/index.php/Quick_Reference_Guide.
45. The Boost community. The Boost C++ Libraries. Retrieved on 08 March. 2012, from http://www.boost.org.