Scientific Programming 21 (2013) 17–42
DOI 10.3233/SPR-130361
IOS Press

A parallel ghosting algorithm for the flexible distributed mesh database

Misbah Mubarak *, Seegyoung Seol, Qiukai Lu and Mark S. Shephard
Scientific Computation Research Center, Rensselaer Polytechnic Institute, 110 8th St, Troy, NY 12180, USA
E-mails: {mubarm, seols, luq3, shephard}@rpi.edu

Abstract. Critical to the scalability of parallel adaptive simulations are parallel control functions including load balancing, reduced inter-process communication and optimal data decomposition. In distributed meshes, many mesh-based applications frequently access neighborhood information for computational purposes which must be transmitted efficiently to avoid parallel performance degradation when the neighbors are on different processors. This article presents a parallel algorithm of creating and deleting data copies, referred to as ghost copies, which localize neighborhood data for computation purposes while minimizing inter-process communication. The key characteristics of the algorithm are: (1) It can create ghost copies of any permissible topological order in a 1D, 2D or 3D mesh based on selected adjacencies. (2) It exploits neighborhood communication patterns during the ghost creation process thus eliminating all-to-all communication. (3) For applications that need neighbors of neighbors, the algorithm can create n number of ghost layers up to a point where the whole partitioned mesh can be ghosted. Strong and weak scaling results are presented for the IBM BG/P and Cray XE6 architectures up to a core count of 32,768 processors. The algorithm also leads to scalable results when used in a parallel super-convergent patch recovery error estimator, an application that frequently accesses neighborhood data to carry out computation.

Keywords: Ghost entities, mesh database, neighborhood communication, massively parallel architectures

1. Introduction

Unstructured mesh methods, like finite elements and finite volumes, effectively support the analysis of complex physical behaviors modeled by partial differential equations over general three-dimensional domains. In many cases, the ability to solve problems of practical interest requires the use of billions of spatial degrees of freedom and thousands of implicit time steps. Problems of such size can only be solved by distributing the mesh on massively parallel computers. As analysis is performed on a massively parallel computer with distributed meshes, consideration must be given to inter-process communication and synchronization to ensure the efficiency of such parallel computations.

In a distributed unstructured mesh, most of the communications are within a set of neighboring processors. That is, the analysis code accesses adjacent mesh entities to compute parameters needed during the computation. Therefore, it is important to cluster elements or mesh entities in a way to minimize the number of adjacent mesh entities belonging to different processors, thus minimizing communication [9]. Even then, specific operations for example, mesh smoothing, calculating error estimate and local geometric reconstruction (e.g. subdivision surfaces) requires data from mesh entities internal to neighboring processors. One means to support providing this data that minimizes communication overheads, is by the addition of copies of the data, referred to as ghost copies, as needed on neighboring processors. The basic purpose of a ghost copy is to localize the data for computation on partition boundaries. The ghosting algorithm provides an application with the complete parallel neighborhood information in a partitioned mesh which can speed-up repeating calculations. If the ghost data becomes invalid due to ghost values change, ghost data must be updated [16]. Some applications like parallel mesh smoothing [11] and parallel error estimation [37] may require a single ghost layer which comprises of mesh entities adjacent to the partition boundary entities. In some cases, applications like local error estimation on high order discretization, or parallel geometric approximation improvement [34], more than one

*Corresponding author: Ms. Misbah Mubarak, Scientific Computation Research Center (SCOREC), Rensselaer Polytechnic Institute, 110 8th St, Troy, NY 12180, USA. Tel.: +1 518 951 0857; E-mail: mubarm@rpi.edu.

1058-9244/13/$27.50 © 2013 – IOS Press and the authors. All rights reserved

layer of ghost entities can be required; the new ghost layer consists of neighboring mesh entities of the previous ghost layer. To support the application requirements, the ghosting algorithm presented in this paper can ghost up-to n ghost layers. In addition to direct unstructured mesh applications, the ghosting methods presented here can be used for other applications where topological adjacency information can be used to coordinate the sharing of neighborhood information. An example of this type under current development are particle-in-cell methods [30,32] where copies of particle data for neighbors is critical.

For distributed meshes, it is useful to define a component which defines both a specification for an API and an abstract data model that specifies the semantics of the data passed through the interface. The advantage of a component based approach is that the focus is on interfaces rather on data structures or file formats. This allows any application using the component to use another implementation of the same component API, as all implementations comply with the same component-based functionality. iMeshP is a parallel unstructured mesh component developed by the Interoperable Technologies for Advanced Peta-scale Simulations (ITAPS) [17], which provides the functionality to create, query and modify a parallel distribution of a mesh. The Flexible distributed Mesh DataBase (FMDB) is an iMeshP compliant distributed mesh management system that provides a parallel mesh infrastructure capable of handling general non-manifold models while effectively supporting parallel adaptive analysis [27,28]. The ghost creation/deletion algorithm presented in this paper is developed for FMDB as part of the iMeshP specification for effectively supporting ghost entity creation, retrieval and deletion.

This paper describes a parallel N-layer ghost creation and deletion algorithm that utilizes neighborhood communication to create up to n ghost layers ($n \geqslant 1$). To the best of our knowledge, this is the first paper that describes the ghost creation and deletion algorithms with a detailed performance analysis on massively parallel architectures up to a core count of 32,768 MPI processes. The contribution of this paper is: (1) It presents an efficient ghost creation and deletion algorithm by utilizing neighborhood communication methodology [23] to reduce the message passing overheads; (2) The algorithm has been shown to scale for massively parallel architectures like Blue Gene/P and Cray XE6 up to a core count of 32,768 processors; (3) The usage of the algorithm has been demonstrated in massively parallel meshing applications i.e. the parallel error estimation procedure with a weak scaling study on a high scale of 32,768 MPI processes. The organization of the paper is as follows: Section 2 provides information about distributed mesh representation which is essential in understanding the ghosting algorithm. Sections 3 and 4 discuss the design and implementation of N-layer ghost creation and deletion algorithms. Section 5 presents strong and weak scaling results of the ghosting algorithm followed by an N-layer efficiency analysis on two massively parallel architectures i.e. Cray XE6 and IBM BG/P up to a core count of 32,768. Section 6 describes the application of ghosting in parallel super-convergent patch recovery (SPR) based error estimation.

2. Distributed mesh representation

The process of ghost creation/deletion is dependent on having a topology-based mesh database that answers basic queries about mesh entities and their topological adjacencies [3]. For the ghost creation/deletion algorithm, we use the Flexible Distributed Mesh Database (FMDB) [27,28]. FMDB is a distributed mesh management system which effectively supports parallel adaptive analysis and makes parallel mesh infrastructure capable of handling general non-manifold models. This section gives a description of FMDB, its abstract data model and meshing terminologies followed by a description of the parallel mesh component, iMeshP.

The mesh is a collection of mesh entities with controlled size, shape and distribution. The relationships of entities of a mesh are defined by topological adjacencies, which form a graph of the mesh [3]. Unstructured meshes are effectively described using a topology based representation in which the members of the hierarchy of topological entities of regions, faces, edges and vertices are defined plus adjacencies that describe how the mesh entities connect to each other. There are many options in the design of mesh data structure in terms of which entities are explicitly represented and which connectivities are stored [27]. If a mesh representation stores all 0 to d dimension entities explicitly, where an entity dimension can be a vertex, edge, face or a region, then it is a full representation. Completeness of adjacency indicates the ability of a mesh representation to provide any type of adjacencies requested without involving an operation dependent on the mesh

size such as the global mesh search or mesh traversal. Regardless of whether the representation is full or reduced, if all adjacency information is obtainable in O(1) time, the representation is complete, otherwise, it is incomplete.

We assume a full and complete representation through out this paper. Implementations with reduced complete representations using all the same overall algorithms are possible, with the addition of some complexities within the mesh database API functions [27].

In a distributed mesh representation, the mesh domain is decomposed into a number of parts, or subdomains where each part holds a subset of a mesh. Each part is treated as a serial mesh with addition of mesh part boundaries for entities that are on inter-part boundaries. The mesh entities on part-boundaries are shared by more than one part. In distributed meshes, each processor holds one or more mesh parts. In this paper, we use the mapping of one mesh part assigned to each MPI process. Due to the evolving nature of adaptive methods, load imbalance can be introduced into the solution process which makes it critical to dynamically load balance the mesh entities. To effectively support mesh adaptation, FMDB supports dynamic entity migration and load balancing procedures [27].

To handle the requirements of distributed memory applications, the ITAPS parallel mesh component, iMeshP, provides a distributed representation [22]. The abstract data model of iMeshP consists of the core concepts and data structures to support distributed meshes i.e.: (1) A mesh partition, which is responsible for mapping entities to parts and parts to processes; (2) Mesh entities are owned by exactly one part where the owner mesh entity has the right of modification; (3) Mesh entities can be classified as an internal entity (an entity not on inter-part boundary), a part-boundary entity (an entity on inter-part boundary which are shared between parts) or a ghost entity (a non-part boundary entity which is not owned by the part). A common interface is built on top of this abstract data model to support distributed mesh operations for example, creating and modifying partitions, create and delete ghost entities, entity migration, determine an entity's ownership status etc. To support the ghost entity creation, retrieval and deletion functionality in FMDB as part of the iMeshP specification, we have designed the ghost creation and deletion algorithm presented in this paper.

Before describing the ghosting algorithms, it is useful to introduce some nomenclature in FMDB.

Mesh nomenclature

M	an abstract model of the mesh.
M_i^d	the ith entity of dimension d in the model M. The entity dimension d of an entity is 0 for a vertex, 1 for an edge, 2 for a face and 3 for a region.
$\{\partial(M_i^d)\}$	set of entities on the boundary of M_i^d.
$\{M_i^d\{M^q\}\}$	set of entities of dimension q that are adjacent to M_i^d. For example $\{M_1^3\{M^0\}\}$ is the set of all vertices that are adjacent to the region M_1^3.
$\mathscr{P}[M_i^d]$	set of part id(s) where M_i^d exists.
$\{M_i^d\{M^q\}_j\}$	The jth entity in the set of entities of dimension q in model M that are adjacent to M_i^d. For example $\{M_1^3\{M^0\}_2\}$ is the second vertex adjacent to region M_1^3.

The union of mesh entities over all the parts define the complete mesh being operated in parallel [28]. Mesh entities can be migrated between parts; that process includes constructing/updating on-part and inter-part mesh adjacencies. Figure 1 represents a 2D mesh distributed on four parts where solid lines show the part boundaries. The vertex M_i^0 is common to four parts and on each part several edges like M_j^1 are common to two parts. To keep the part boundary mesh entities synchronized with their copies, mesh entities on part boundaries must keep track of their duplicates. One way to do this is to store remote copy and remote part information for each part boundary entity.

Definition 1 (Remote part). A part where an entity is duplicated.

For example in Fig. 1, M_j^1 is owned by part P_1 and it has a remote copy on part P_0. M_j^1 will keep information on part P_1 about the remote part P_0 where it has been duplicated. Parallel methods are provided to determine inter-part adjacency information for mesh entities (faces, edges and vertices) on the inter-part boundaries. These parallel methods develop communication links among parts to provide remote copy and ownership information of mesh entities that are duplicated from other parts.

Definition 2 (Remote copy). Refers to memory location of a mesh entity duplicated on a remote part p.

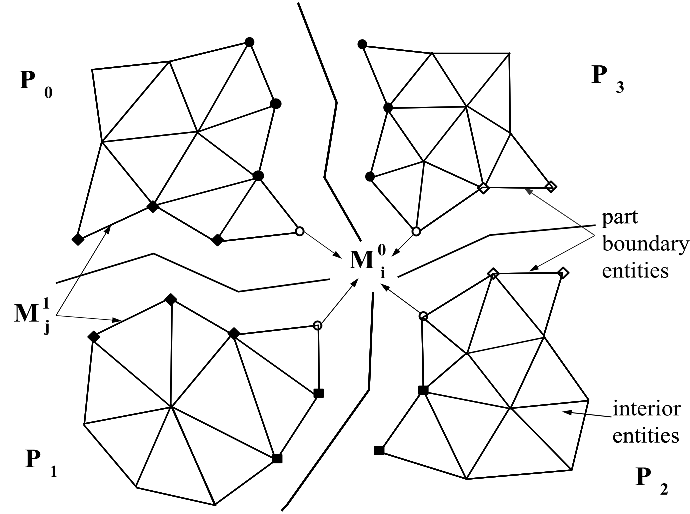


Fig. 1. Distributed Mesh on four parts.

The memory location is the address where an entity is duplicated on a remote part. For example, in Fig. 1, M_i^0 is duplicated on parts P_0, P_1, P_2 and P_3. Part P_0 owns entity M_i^0 and M_i^0 stores the memory addresses of its remote copies on rest of three parts. This keeps the owner entity M_i^0 on part P_0 synchronized with its remote copies on other parts.

Mesh entities are owned by exactly one part where ownership is the right to modify. Ownership of an entity can be changed during the course of computation due to re-partitioning of a mesh or due to migration operations. For entities on part boundaries, the entity will exist on several parts, one of which is designated the owner. Within a part, a mesh entity can have one of the three classifications.

- Internal Entity: An owned entity that is not on a part boundary. For example, in Fig. 1, interior entities are labeled. An interior entity does not have a remote copy on other parts.
- Boundary Entity: A shared entity on a part boundary. For example, in Fig. 1, M_i^0 is a part boundary vertex which is owned by P_1 and has a remote copy on other parts.
- Ghost Entity: A read-only copy of a non-owned entity that is not a shared part boundary entity. For example, Fig. 2 shows ghost faces M_i^2 which are created after the one layer ghost creation procedure is applied.

Definition 3 (Neighboring parts). Part A neighbors Part B if Part A has remote copies of entities owned by part B or if part B has remote copies of entities owned by part A.

For example, in Fig. 1, the neighboring parts of P_0 are P_1, P_2 and P_3 as M_i^0 which is owned by P_0 has remote copies on P_1, P_2 and P_3. Within a part, the term "copy" of an entity refers to either ghost entity or a part boundary entity owned by a remote part.

It is also useful to define the notion of a partition model entity and partition classification. A partition model entity groups the mesh entities residing on the same part and the partition classification provides a mapping of the mesh entities to their partition model entity.

Definition 4 (Partition model entity). A partition model entity, P_i^d, represents a group of mesh entities of dimension d that reside on the same mesh part.

Definition 5 (Partition classification). The unique association of mesh topological entities of dimension d, M_i^d to the partition model entity P_i^d.

3. Design of ghosting algorithm

The ghost creation and deletion algorithm can be effectively used to reduce communication costs in applications that need access to neighborhood data during computation [20]. Previous work [2,7,19,35] has been done to create ghost data through message-passing. However, most of these approaches use MPI collective

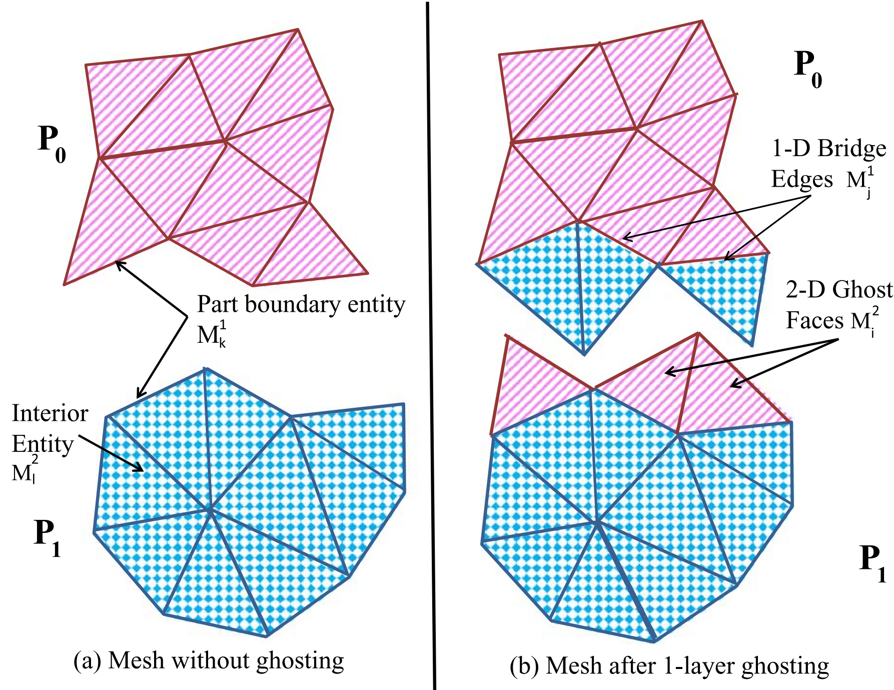


Fig. 2. A distributed mesh on two parts. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130361.)

operations to exchange ghost messages and others do not present the scaling data of the ghosting algorithms. In general, the ghost creation and deletion process can be applied to support any parallel meshing applications whose operations rely heavily on inter-process data exchange and movements to reduce communication overheads. In the areas of finite element simulations, aside from the parallel error estimation application, ghosting can be used with parallel mesh smoothing, and geometric reconstruction procedures, such as subdivision surfaces, to help eliminate extensive inter-process communications.

The design requirements for the N-layer ghost creation/deletion algorithm follow those of the iMeshP specification [16]. Ghost entities are copies of additional, non-part boundary entities which are requested by an application to enable efficient computation. A ghost entity is specified through a bridge entity and a ghost entity can be either a mesh region, face or an edge. Ghost regions are used to support a number of mesh-based applications, two common examples are: (1) Error estimation procedures like parallel super-convergent patch recovery presented in Section 6; (2) Parallel mesh smoothing operations to synchronize smoothing of part boundary vertices. The number of layers in the ghosting process are measured from the inter-part boundary. The first layer of ghosts comprises of entities that are directly adjacent to the part boundary entities. Entities in the next ghost layer comprise of neighbors of the previous ghost layers through a bridge entity. Some applications like interpolation subdivision surface scheme [34], in which the whole patch of the mesh entities to be subdivided is distributed on parts, require at-least two ghost layers. This section describes the design aspects of the N-layer ghost creation/deletion algorithm.

3.1. Input and output parameters of ghost creation process

Before describing the input parameters for ghost creation, it is useful to describe the concepts of ghost and bridge entities.

Definition 6 (Ghost entity vs. bridge entity). If M_j^b is a bridge entity and $\{M_j^b\{M^g\}_i, b < g\}$, is not a null set, then every entity M_i^g in the set is a ghost candidate.

An application must specify the following parameters to initiate the ghost creation process:

- *Ghost type "g"*: g is the dimension of the ghost entities where the ghost dimension can be a region, face or an edge in a 3D mesh.

- *Bridge type "b"*: b is the dimension of the bridge entities where the bridge dimension can be a face, edge or a vertex in a topological mesh representation.
- *Number of layers "n"*: n is the number of layers of ghost entities. Layers are measured from inter-part boundary of the mesh.

Bridge entities are on the boundary of higher order mesh entities that are to be ghosted. Thus, the lowest possible dimension of a ghost entity can be an edge in which case the bridge dimension will be a vertex. For example, Fig. 2 shows 1 ghost layer with bridge edges (M_j^1) and ghost faces (M_i^2) which makes the ghost dimension, bridge dimension and number of layers as 2, 1 and 1 respectively. The ghost creation algorithm takes the ghost dimension, bridge dimension and number of layers to be created and outputs a ghosted mesh with ghost entities of *ghost type "g"* created using bridge entities of *bridge type "b"* with *"n"* layers of ghost entities.

3.2. Ownership of ghosts

The iMeshP data model defines rules about the amount of information that an implementation must manage. To avoid excessive communication between parts, whenever a ghost entity is created, the original entity whose ghost is created, is designated as the owner entity. The owner entity must locally store information about all its ghost copies that exist on other parts. The ghost copy must also store information about its owner entity and the owning part. This provides a mechanism to keep the owner entity synchronized with their ghost copies on other parts and vice versa [6,16]. By storing this information locally, the following queries can be answered without involving remote communication between parts (i) the ownership information of a ghost entity or (ii) the information about the ghost copies of an original entity.

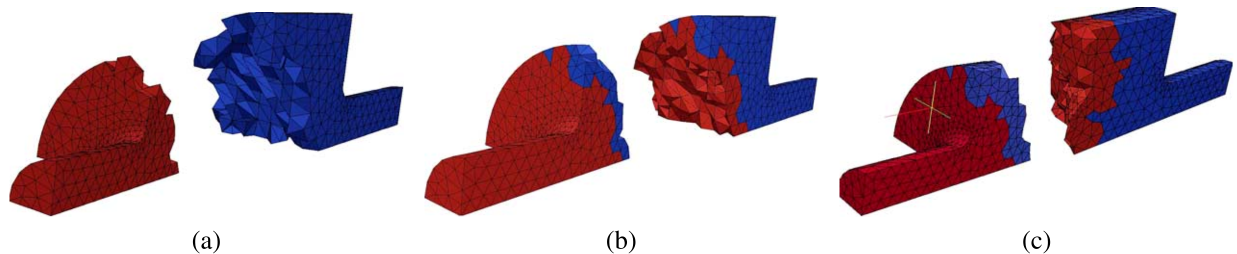
3.3. Multiple ghost layers

Another feature of the ghosting algorithm is the ability to support multiple layers. When multiple layers are requested by an application, the ghosting process starts with the first (innermost) layer of ghosts adjacent to the part boundary. After collecting entities for the first layer, it collects entities for the second layer by treating the first layer of ghosts as part of the mesh for the second layer. For multiple layers, it marks all bridge and ghost entities already processed for the previous layers. Care is taken that the same element is not added as a ghost again, at this point the algorithm is required to create only single copies of a ghost entity.

Figure 3(a) demonstrates an initial 2-part mesh before the application of ghost creation procedure. Figure 3(b) and (c) shows the same 2-part ghosted mesh with one and two layers of ghost regions (bridge vertices) surrouding the part boundaries.

3.4. Communication pattern

In shared memory systems, data read/write operations are straight-forward as the processes share the same address space whereas in distributed systems, each process has its own address space and data communication is performed through message-passing. Adaptive meshes can have billions of elements and the analysis procedure for such large meshes may require a million or more compute cores. In the past, most of the massively parallel architectures such as IBM Blue Gene series [15], Cray series (like Cray XT5 and Cray XE6 [5]) have effectively supported distributed memory paradigm. Recently with systems like IBM Blue Gene/Q [12] and Cray XC30 [21], the possibilities of using a hybrid paradigm comprising of threads and message-passing are being evaluated. We are currently working to support a hybrid paradigm for FMDB. The usage of hybrid memory paradigm in FMDB will



(a) (b) (c)

Fig. 3. One and two layer ghost creation on an electromagnetic mesh divided in two parts. (a) Partitioned mesh without ghosting. (b) Mesh after 1-layer of ghost regions. (c) Mesh after 2-layers of ghost regions. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130361.)

eliminate the overhead of duplicating data within the parts residing on the same compute node. With hybrid paradigm in place, the ghosting algorithm will only be required for part boundaries between parts that reside on different MPI process. We further anticipate that there will be no need of ghost creation/deletion among parts that reside on the same compute node due to shared memory access. In this paper, we evaluate the ghosting algorithm with distributed memory paradigm as it is effectively supported by most massively parallel architectures.

In distributed mesh applications, most of the inter-part communication involves neighboring MPI processors. The ghost creation and deletion algorithms is based on a distributed memory paradigm [1,10] where ghost entities are created by sending ghost creation messages to its neighboring parts. For large meshes having billions of elements, the number of ghost creation messages can be considerably large which may cause increased latency if the communication is not optimized. Communication during ghost creation can be optimized by avoiding unnecessary collective communication operations and by buffering small messages. For this purpose, the ghosting algorithm uses IPComMan [24], a general purpose communication package built on top of MPI that provides neighborhood communication by eliminating large number of small message through message packing. For readers interested in the distinguishing aspects of the neighborhood communication and message packing in parallel meshes, details are presented in [23,24]. Previous experiments concerning the scales of meshes on parallel machine are also discussed in this work.

As a mesh entity of dimension > 0 is bounded by lower dimension entities, these lower dimension entities are also carried along for ghost creation at destination parts. One tempting approach is to carry along all the lower dimension downward adjacent entities of a ghost. For example, if the ghost dimension is region, then its downward adjacent faces, edges and vertices can also be carried to destination parts. Doing so introduces extra communication overhead as the number of ghost creation messages will increase significantly. To avoid this situation, the ghosting algorithm only takes the top-dimension entity to be ghosted and its downward adjacent vertices to the destination part. At the destination parts, local copies of missing adjacencies are created using vertex information.

3.5. *Mesh entity and ghost information*

A mesh entity in FMDB is a data structure which stores information such as its geometric classification, partition classification and remote copy information. To relate the ghost information with the mesh entity data structure, we store the ghost information in an independent data structure and link it to the mesh entity data structure using an *index number*. This way, a mesh entity's ghost information is retrieved in O(1) time using the *ghost index number*.

Figure 4 shows the class diagram that describes the relationship of the ghost information with the mesh entity data structure. A mesh entity can be a vertex, edge, face or a region. A mesh vertex stores its coordinates information (coords) and upward adjacency information (up_adj). An edge and a face both store upward and downward adjacency information (up_adj and down_adj resp.). Faces and regions also store their topology information (topo). A region only stores downward adjacency information (down_adj) as it has the highest topological dimension. A mesh entity may or may not have ghosting information but the ghosting information is always related to a mesh entity.

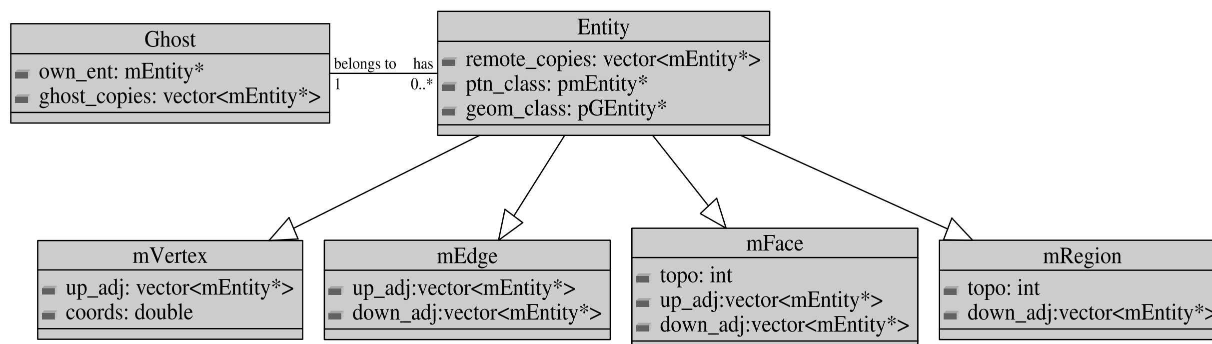


Fig. 4. Class diagram of mesh entity and ghosting information.

4. Implementation of ghosting algorithm

The computational model of the ghost creation and deletion algorithm follow the FMDB model described in Section 2. This section describes the algorithms used in the implementation of N-layer ghost creation and deletion. It also discusses the parallel aspects of the algorithm i.e. mapping of parts to MPI processes and how the global coordination is performed between the parts.

4.1. *Overview of ghost creation and deletion algorithms*

The input to the ghost creation algorithm uses a distributed mesh representation in which the mesh is divided into mesh parts. Each mesh part is treated as a serial mesh with addition of part boundaries for entities that lie on the inter-part boundary. Each part is mapped to a single MPI process. The communication between the parts is done through message-passing. Algorithm 1 presents the major steps required in the ghost creation algorithm. The computation in Steps 1 and 2 is performed locally on a part (resp. MPI process). Step 3 performs nearest-neighbor communication with neighboring parts using IPComMan neighborhood communication package [23] to eliminate duplicate remote entities. Step 4 also exploits the neighborhood communication patterns of IPComMan to create ghost entities on neighboring parts. The ghost deletion algorithm performs local computation to delete any existing ghost entities. It does not require communication among parts for deleting ghost entities.

As ghost entities are created to localize the non part-boundary data on neighboring parts, ghost entities may become outdated after mesh modification operations are performed. In such a case, following mesh modification, synchronization may be performed among parts to ensure that the distributed mesh is consistent and valid across all parts. During mesh synchronization, ghost entities that require an update, may be deleted using the ghost deletion algorithm and then re-created through the ghost creation procedure. This ensures that the ghost data stays up-to-date after mesh modification.

4.2. *Ghost creation algorithm*

The ghost creation algorithm *Create_Ghosts* creates ghost entities in a distributed mesh given the dimension of ghost and bridge entities (g and b respectively) and the number of ghost layers (n). The overall procedure for ghost creation comprises of the following steps:

1. Collect entities of given ghost dimension "g" adjacent to part boundary bridge entities of dimension "b" and determine the destination parts of the "g" dimensional entities.
2. If the specified number of ghost layers is more than 1, process next ghost layer such that the ghost entities in previous layers are not added again.
3. For part boundary entities that exist on multiple parts, more than one part can collect the same part boundary entity (original entity or its remote copy) for the same destination part. One round of communication is carried out to eliminate any duplicate ghost creation messages.
4. Exchange ghost entity information among parts, create ghost entities on destination parts and update ghost owner entities with ghost copy information.
5. Store ghost information rule consisting of g, b and n in the part so that the ghosting information can be retrieved later during ghost deletion process or mesh synchronization.

Figure 5(a) and (b) illustrates the example 2D partitioned mesh to be used to demonstrate the working of ghost creation and deletion algorithm throughout this section.

Algorithm 1 shows the steps used for ghost creation given the parameters g, b and n. During the ghost creation process, the entities collected for ghosting are maintained in a vector *entitiesToGhost* where *entitiesToGhost*$[1 \ldots n]$ contains entities to be ghosted at each layer. The downward adjacent vertices that are required for ghost creation are collected in *entitiesToGhost*$[0]$. Each part maintains a separate *entitiesToGhost*$[0 \ldots n]$ vector and every entity collected for ghosting has a data structure *ghostParts*$[M_i^g]$ which is mapped to M_i^g. To keep track of the destination parts of a ghost entity, the data structure *ghostParts* holds the destination part IDs of M_i^g. This way, the destination part of a ghost entity can be efficiently retrieved in O(1) time without carrying out any form of search operation.

For simplicity of notation, we use (1) P_{remote} to denote a remote part (2) $\mathscr{R}[M_i^d]$ to denote the set of remote copies of an entity M_i^d.

4.2.1. *Step* 1: *Ghost collection for first layer*

Algorithm 1, Step 1 collects ghost candidates adjacent to part boundary bridge entities in the first layer.

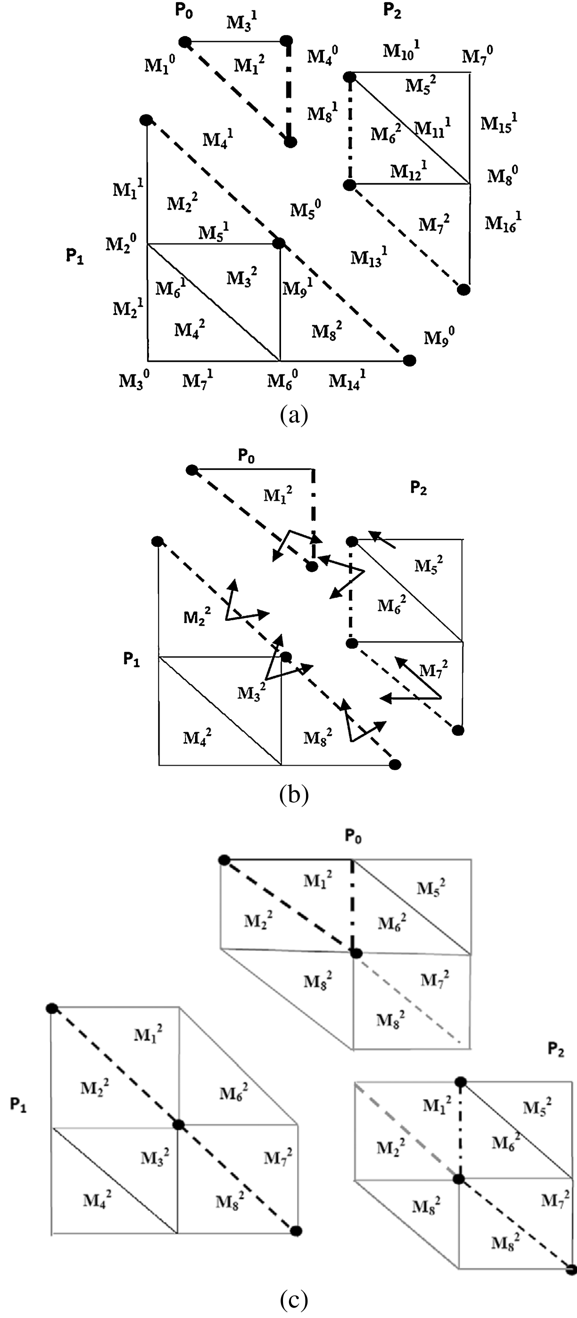


Fig. 5. Example of 2D ghost creation procedure with 1-layer of ghost faces. (a) Initial mesh. (b) Mesh during 1-layer ghosting. (c) Mesh after 1-layer ghosting.

Definition 7 (Destination part rule for 1st layer ghosts M_i^g). If $\{M_i^g\{M^b\}_j\}$, $b < g$, and $p \in \mathscr{R}[M_j^b]$, $p \notin \mathscr{R}[M_i^g]$ then $p \in$ *ghostParts*$[M_i^g]$.

An entity, M_i^g is collected for ghost creation using Definition 7. If one of the bridge entities M_j^b, that is on the boundary of ghost candidate M_i^g, has a copy on a remote part p but the ghost candidate M_i^g has no copy on p then M_i^g will be ghosted on remote part p.

Algorithm 2 collects ghost entities for the first layer. For the example mesh (Fig. 5), Table 1 shows the contents of *entitiesToGhost* after Step 1. Contents of *ghostParts*$[M_i^d]$ are enclosed in brackets along with each entity in Table 1.

4.2.2. Step 2: Process next layer

If the number of layers specified for ghosting is more than one, subsequent ghost layers are added after the first layer. For the next layer, entities of dimension g that share a bridge entity with any $M_i^g \in$ *entitiesToGhost*$[n-1]$ are collected for ghost creation. To make sure that the same entity is not added more than once for ghosting, a visited tag is set for each M_i^g marked for ghosting (see Algorithm 2, Step 2). Setting visited tags for bridge entities cuts down the computation time of the algorithm as the same entity is not processed again when computing subsequent ghost layers.

For an entity M_k^g in the second to nth layer, destination parts are decided using Definition 8 where an entity M_k^g that shares a bridge entity M_i^b with one or more ghost candidates of the previous ghost layer ($M_l^g \ldots M_m^g$) is also ghosted to the destination parts of those ghost candidates ($M_l^g \ldots M_m^g$). Algorithm 3 gives the pseudo code for Nth layer entity collection that decides destination parts using Definition 8. As an example, the 1-layer ghost creation in Fig. 5 is extended for two layers in Fig. 6 which shows ghost entity collection for two layers. The second layer bridge vertices are depicted by blue circles around them.

Definition 8 (Destination part rule for nth ($n > 1$) layer ghost M_k^g). If $\{M_k^g\{M^b\}_i\}$ where $\{M_i^b\{M^g\}_l\} \ldots \{M_i^b\{M^g\}_m\}$ s.t. $\{M^g\}_l \ldots \{M^g\}_m \in$ *entities ToGhost*$[n-1]$, then *ghostParts*$[M_k^g]$ = *ghost Parts*$[\{M^g\}_l] \cup \cdots \cup$ *ghostParts*$[\{M^g\}_m]$.

For example, in Fig. 6, as M_2^2 (first layer ghost) shares a common bridge entity M_2^0 with M_4^2 (second layer ghost), the *ghostParts*$[M_2^2]$ will be added to *ghostParts*$[M_4^2]$. Table 2 gives the contents of vector *entitiesToGhost* after Step 2. Contents of *ghost Parts*$[M_i^d]$ are enclosed in brackets along with each entity in Table 2.

Algorithm 1. Create_Ghosts(M, g, b, n)

Data: distributed mesh M, ghost dimension g, bridge dimension b, number of ghost layers n

Result: Creates ghosts on the distributed mesh M

```
begin
    /* Step 1: collect ghost entities in the first layer */
    getGhostEnts(M, g, b, n, entitiesToGhost);
    /* Step 2: Process next ghost layer if n > 1 */
    for lyr ← 2 to n do
        processNLayers(M, g, b, lyr, entitiesToGhost);
    end for
    /* Step 3: Do one round of communication to eliminate duplicate remote
       entities */
    removeDuplicateEnts(M, entitiesToGhost[0]);
    /* Step 4: Exchange entities and update ghost information */
    exchangeGhostEnts(M, g, entitiesToGhost);
    /* Step 5: Store ghost information in the part */
    ghostRule → [g, b, n]
end
```

4.2.3. *Step* 3: *Eliminate duplicate entities*

A part boundary entity exists on multiple parts so many parts can collect it for ghost creation to the same destination part. For this reason, Table 1 and Table 2 have duplicate vertex entries marked by a *. Once all ghost entities and their downward adjacent vertices are collected for ghosting, one round of neighborhood communication needs to be performed among the process neighborhood to eliminate duplicate ghost creation messages. The communication step introduces minimal overhead as: (i) It is only performed for the inner most ghost layer next to the part boundary as this ghost layer may have duplicate part boundary entities; (ii) It utilizes IPComMan neighborhood communication which restricts the communication to a set of neighboring processors. Algorithm 4 gives the pseudo code for eliminating duplicate ghost creation messages.

One round of communication is performed in Algorithm 4 to determine duplicate ghost creation messages. If multiple parts intend to send ghost creation message for M_i^d to the same destination part p, after this step only the part having minimum part id will send the ghost creation message of M_i^d to p. For the example mesh in Fig. 6, both P_0 and P_1 collect M_1^0 for sending it to P_2 but after Algorithm 4 eliminates duplicate entities, only P_0 sends M_1^0 to P_2. Table 3 shows the contents of *entitiesToGhost* after eliminating duplicate entries for remote copies. Duplicate entities with a $*$ on them are no more there.

4.2.4. *Step* 4: *Exchange entities and update ghost copies*

This step creates ghost entities on destination parts by exploiting neighborhood communication pattern and message packing features of IPComMan [23]. Mesh entities are exchanged by first exchanging ghost vertices and then the ghost entities of dimension g. Step 4 of Algorithm 1 creates ghost entities on destination parts. Algorithm 5 is the pseudo-code that exchanges the entities contained in *entitiesToGhost*. After creating ghosts on the destination parts, ghost entities are assigned their owner entity and owning part ID to synchronize them with their owner. Entities owning the ghosts also receive back information about newly created ghosts based on which they can keep a track of their ghost copies. Algorithm 5 step 5.1 sends the message to the destination parts to create new ghost entities. For each M_i^d collected for ghost creation, each part which sends a message composed of the address of M_i^d on the local part p and the information of M_i^d necessary for entity creation, which consists of

- Owner entity information (to notify the owner about the ghost entity created).
- Owner part information.

Algorithm 2. getGhostEnts(M, g, b, n, entitiesToGhost)

Data: distributed mesh M, ghost dimension g, bridge dimension b, number of layers n, entitiesToGhost

Result: Collect first-layer entities to ghost in container *entitiesToGhost*

```
begin
  /* Collect entities-to-ghost and their downward adjacent vertices     */
  /* Step 1 For every part boundary bridge entity M_i^b, check its upward
     adjacent entity M_j^g for ghosting                                  */
  foreach M_i^b on part boundary do
    foreach M_j^g ∈ {M_i^b{M^g}} do
      /* Step 1.1 If M_j^g does not exist on a remote part p where M_i^b
         exists on p, collect M_j^g for ghosting on p                    */
      foreach part ID p ∈ R[M_i^b] do
        if p ∉ R[M_j^g] then
          insert M_j^g in entitiesToGhost[g];
          insert p in ghostParts[M_j^g];
        end if
      end foreach
      /* Step 2: Mark M_j^g as visited (if multiple ghost layers)        */
      if n > 1 then
        set M_j^g ← visited = true;
      end if
    end foreach
    /* Step 3: Mark the bridge entity as visited (if multiple ghost
       layers)                                                           */
    if n > 1 then
      set M_i^b ← visited = true;
    end if
  end foreach
end
```

Table 1

Contents of vector *entitiesToGhost* after Step 1 “Collect ghost entities” of ghost creation algorithm

	P_0	P_1	P_2
entitiesToGhost[0]	$M_1^0\{2\}$, $M_4^0\{1\}$	$M_1^0\{2\}^*$, $M_2^0\{0,2\}$, $M_9^0\{0\}$, $M_6^0\{0,2\}$	$M_4^0\{1\}^*$, $M_7^0\{0\}$, $M_8^0\{0,1\}$, $M_9^0\{2\}^*$
entitiesToGhost[1]	$M_1^2\{1,2\}$	$M_3^2\{0,2\}$, $M_8^2\{0,2\}$, $M_2^2\{0,2\}$	$M_6^2\{0,1\}$, $M_5^2\{0\}$, $M_7^2\{0,1\}$

- Geometric classification information (so that the created ghost entity also has the same geometric classification).
- Topology information (whether the entity is a vertex, line segment, triangle, quadrilateral, hexahedron, tetrahedron, prism or pyramid).
- Vertices information (if non-vertex).
- Attached data (if any).

Every non-vertex entity (edge, face or region) carries along the vertex information as it is required to create a new non-vertex entity on the destination part. For example, in Fig. 6, when M_2^2 is sent to P_2, it takes

Algorithm 3. processNLayers(M, g, b, lyr, entitiesToGhost)

Data: M (distributed mesh), g (ghost dimension), b (bridge dimension), n (number of ghost layers), entitiesToGhost

Result: Collects entities for ghosting nth layer

```
begin
  /* Step 1 Process n−1th layer to get nth layer                              */
  foreach M_i^g ∈ entitiesToGhost[n − 1] do
    /* Step 1.1 For M_i^g, get its downward adjacent bridge entities           */
    foreach M_k^b ∈ {M_i^g{M^b}} do
      /* Step 1.2 If M_k^b is visited, continue with next bridge entity        */
      if M_k^b ← visited == true then
        continue;
      end if
      /* Step 1.3 For the downward adjacent bridge M_k^b, get upward
         adjacent entities of dimension g.                                     */
      foreach M_j^g ∈ {M_k^b{M^g}} do
        /* Step 1.4 If the upward adjacent entity M_j^g is not visited,
           update its ghostParts and collect it for ghosting                   */
        if M_j^g ← visited == true then
          continue;
        end if
        insert M_j^g in entitiesToGhost[n];
        ghostParts[M_j^g] ← ghostParts[M_j^g] ∪ ghostParts[M_i^g];
        /* Step 1.5 Set visited tag M_j^g so that it is not processed
           again                                                               */
        M_j^g ← visited = true;
      end foreach
    end foreach
    /* Step 1.6 Set visited tag of bridge entity M_k^b so that it is not
       processed again                                                         */
    M_k^b ← visited = true;
  end foreach
end
```

along address of M_2^2 on P_1 (owner entity information), P_1 (owner part information), geometric classification, topology information (triangle), address of M_1^0, M_2^0 and M_5^0 (downward adjacent vertices) on the destination part P_2.

Step 4.2 creates a new entity M_i^d on P_i for the message A received on P_{remote} (sent in Step 4.1). The newly created ghost entity updates information about its owner entity and the part where owner entity exists. This information is extracted from the message received. For example, M_2^2 is created on P_2 using the topology information, vertex coordinates and geometric classification from the message A. P_2 then adds the address of M_2^2 on P_0 as its owner entity and sets P_0 as the owning part id of M_2^2.

Step 4.3 sends back the information of the newly created ghost entity to its owner part. For example, in Fig. 6, when M_2^2 is created on P_2, message B is sent back to P_0 with the ghost information of M_2^2 on part P_2.

In step 4.4, the owner entities of ghost objects receive back the ghost information and update their ghost

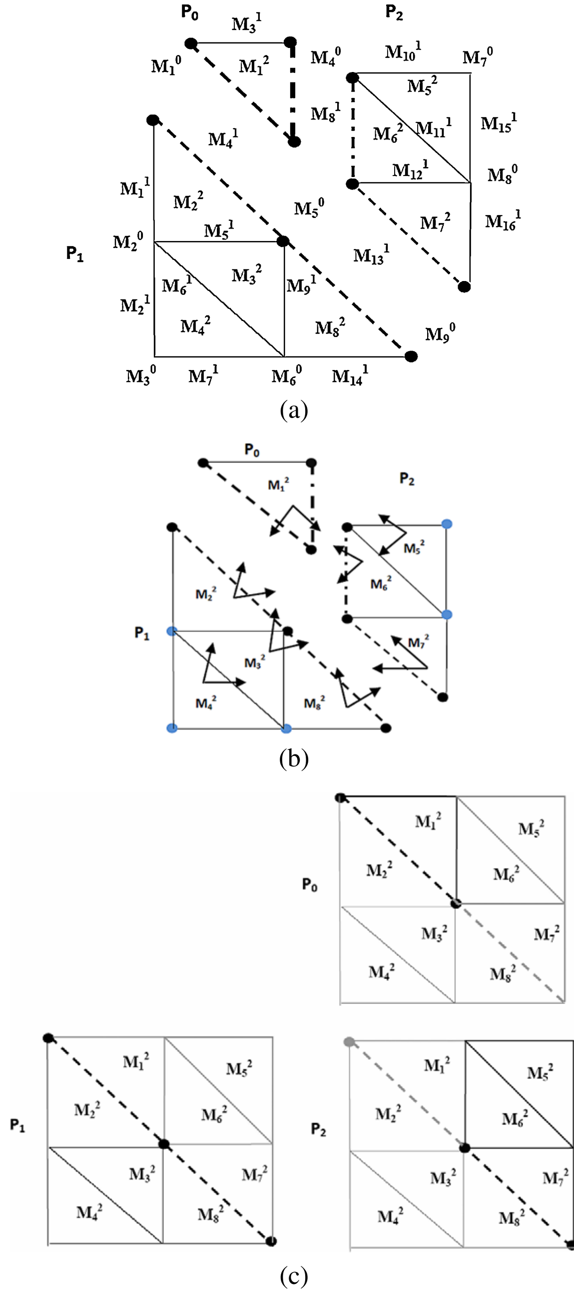


Fig. 6. Example of 2D ghost creation procedure with 2 layers of ghost faces. (a) Initial mesh. (b) Mesh during 2-layer ghosting. (c) Mesh after 2-layer ghosting. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130361.)

copies. For example, in Fig. 6, after 2-layer ghosting process, M_2^2 on P_1 will have the entries [(address of M_2^2 on P_0, P_0), (address of M_2^2 on P_2, P_2)] as its ghost information.

4.2.5. *Store ghost rule*

Ghost information is stored in the part so that the ghosts that becomes outdated due to mesh modification can be restored whenever the mesh is synchronized. Step 5 stores the ghost rule composed of $[g, b, n]$ in the part. For the example mesh in Figs 5 and 6, every part will store the ghost information $[2, 0, 2]$.

4.3. *Ghost deletion algorithm*

The ghost deletion procedure takes the distributed mesh as an input parameter and removes all ghosting information within the mesh. As the ghost entities are stored along with other mesh entities, one tempting approach for ghost deletion is to traverse the entire mesh in search for ghost entities. However, the ghosting algorithm stores all ghost information in a separate data structure within the part. The ghost deletion algorithm avoids traversing the entire mesh by accessing and deleting ghost entities in constant time by utilizing the ghost information stored in the part. Algorithm 6 is the pseudo code of the N-layer ghost deletion algorithm. *entitiesToRemove*$[0 \ldots g]$ is an array that holds the ghost entities and their downward adjacent entities that are to be deleted. Step 1 and 2 of the algorithm checks the ghosting information of the entity.

- If the entity M_j^g is a ghost itself, the algorithm collects it for deletion. The downward adjacent entities of M_j^g are collected for deletion if they were created during ghost creation. Care is taken such that any part boundary downward adjacent entities are not deleted.
- If the entity is owner of ghosts, then the algorithm clears its ghosting information.

Step 3 of the algorithm deletes the ghost entities from the mesh. It starts from the ghost dimension entities first as removing lower dimensional entities earlier can create dangling references for its upward adjacencies.

5. Performance results

This section demonstrates the strong scaling of the ghosting procedure executed on very large meshes. The ghost creation and deletion algorithm is applied to a mesh with 133 million elements on a patient specific abdominal aortic aneurysm (AAA) model (see Fig. 7).

Table 2

Contents of vector *entitiesToGhost* after Step 2 "Process next layer" of ghost creation algorithm

	P_0	P_1	P_2
entitiesToGhost[0]	$M_1^0\{2\}$, $M_4^0\{1\}$	$M_1^0\{2\}^*$, $M_2^0\{0,2\}$, $M_3^0\{0,2\}$, $M_6^0\{0,2\}$, $M_9^0\{0\}$	$M_4^0\{1\}^*$, $M_7^0\{0,1\}$, $M_8^0\{0,1\}$, $M_9^0\{0\}^*$
entitiesToGhost[1]	$M_1^2\{1,2\}$	$M_3^2\{0,2\}$, $M_8^2\{0,2\}$, $M_2^2\{0,2\}$	$M_5^2\{0\}$, $M_6^2\{0,1\}$, $M_7^2\{0,1\}$
entitiesToGhost[2]		$M_4^2\{0,2\}$	$M_5^2\{1\}$

Algorithm 4. removeDuplicateEnts(M, entitiesToGhost(d))

Data: M distributed mesh, entitiesToGhost[d]

Result: Eliminates duplicate ghost entity creation messages

```
begin
  /* Step 1: For all part boundary entities collected in
     entitiesToGhost[d], send a message to the parts having their remote
     copies                                                              */
  foreach M_k^d ∈ entitiesToGhost[d] do
    /* If M_k^d is not a part boundary entity, proceed with the next entity
       */
    if M_k^d is not on part boundary then
      /* Continue with next M_k^d ∈ entitiesToGhost[d]                   */
      continue;
    end if
    foreach destId ∈ ghostParts[M_k^d] do
      foreach p ∈ R[M_k^d] do
        send message A(address of M_k^d on p, destId) to p;
      end foreach
    end foreach
  end foreach
  /* Step 2: Receive message from other parts that are sending ghost
     creation message of the same part boundary entity to destination
     part 'destId'                                                       */
  while part P_remote receives message A from part p(address of M_k^d on p, destId) do
    /* Step 3: Check if the remote part is also sending M_k^d to the same
       destination destId.                                              */
    if destId ∈ ghostParts[M_k^d] & ID(p) < ID(P_remote) then
      /* Step 4:Part P_remote with minimum part id will send M_k^d to destId  */
      remove M_k^d from entitiesToGhost[d];
    end if
  end while
end
```

The mesh is obtained through anisotropic mesh adaptation of a mesh obtained from previous adaptive cycles [26,36].

The scaling studies are applied on two massively parallel architectures, Hopper, a Cray XE6 [5,14] at National Energy Research Scientific Computing Cen-

Table 3

Contents of vector *entitiesToGhost*[0] after Step 3 "Eliminate duplicate entities" of ghost creation algorithm

	P_0	P_1	P_2
entitiesToGhost[0]	$M_1^0\{2\}$, $M_4^0\{1\}$	$M_2^0\{0,2\}$, $M_3^0\{0,2\}$, $M_6^0\{0,2\}$, $M_9^0\{0\}$	$M_7^0\{0,1\}$, $M_8^0\{0,1\}$

Algorithm 5. exchangeGhostEnts(M, g, entitiesToGhost)

Data: M (distributed mesh), entitiesToGhost,n (number of ghost layers)
Result: Create ghost entities on destination parts and notify back their owners
begin
  `/* Step 4: Create ghost entities on destination parts for all entitites in` *entitiesToGhost* `*/`
  foreach $d \leftarrow 0$ *to* n **do**
    `/* Step 4.1: Send message with ghost information */`
    foreach $M_i^d \in$ *entitiesToGhost*$[d]$ **do**
      foreach *part ID* $p \in$ *ghostParts*$[M_i^d]$ **do**
        send message A(address of M_i^d on p, owner of M_i^d, information of M_i^d) to p;
      end foreach
    end foreach
    `/* Step 4.2: Receive message with ghost information */`
    while P_{remote} *receives message A(address of* M_i^d*, owner of* M_i^d*, information of* M_i^d*) from part* p **do**
      create M_i^d with information of M_i^d in message A;
      $P_{own} \leftarrow$ owning part of M_i^d;
      `/* Step 4.3:Notify the owner about ghost information */`
      send message B(address of M_i^d on P_{remote}, address of M_i^d on P_{own}) to P_{own};
    end while
    `/* Step 4.4: update ghost copy information on owner part */`
    while P_{own} *receives message B(address of* M_i^d *on* P_{remote}*, address of* M_i^d *on* P_{own}*) from* P_{remote} **do**
      save the address of M_k^d on P_{own} as for ghost copy on P_{remote};
    end while
  end foreach
end

ter and Shaheen, an IBM Blue Gene/P [15] at King Abdullah University of Sciences and Technology. The Blue Gene/P is a 32-bit architecture with four 850-MHz PowerPC 450 CPUs per node, with three cache levels, 4GB DDR2 SDRAM per node. It has five different networks: 3D torus with 425 MBps in each direction, global collective with 850 MBps, global barrier and interrupt, JTAG, and 10 Gigabit Ethernet (optical). The Blue Gene/P's MPI implementation uses the Deep Computing Messaging Framework (DCMF) as a low level messaging interface. It also has a 3D torus topology for point-to-point communication and its MPI implementation supports three different protocols depending on the message size.

Hopper has 6384 nodes where each node is configured with two twelve core AMD Magny Cours 2.1 GHz processors per node. It has a 32 GB or 64 GB DDR3 1333 MHz memory per node. Each core has its own L1 and L2 caches with 64 GB and 512 KB resp. There is a 6 MB L3 cache shared between every 6 cores on the Magny Cours processors. Hopper compute nodes are connected via a custom high-bandwidth, low latency Gemini network with a 3D torus topology which has a routing capacity of 168 GB/s and a bandwith of 9.8 GB/s per Gemini chip. The Gemini network efficiently supports MPI with millions of MPI messages per second and has an inter-node latency on the order of 1 microsecond. The peak GFlop

Algorithm 6. deleteGhostEnts(M)

Data: a ghosted mesh M

Result: Deletes all ghost entities in the mesh

begin
- `/* Step 1: Collect the top-level ghost entity and its downward adjacent entities (if they are not part boundary entities) */`
- **foreach** $M_j^g \in$ *ghost information stored in the part* **do**
  - **if** $M_j^g \leftarrow$ *isGhost* **then**
    - *entitiesToRemove*$[g] \leftarrow M_j^g$;
    - **foreach** $d \leftarrow 0$ *to* g **do**
      - **foreach** $M_k^d \in \{M^d\{M_j^g\}\}$ *where* $d < g$ **do**
        - **if** $M_k^d \leftarrow$ *isGhost* **then**
          - *entitiesToRemove*$[d] \leftarrow M_k^d$;
        - **end if**
      - **end foreach**
    - **end foreach**
  - **end if**
  - `/* Step 2: If the entity is owner of a ghost, clear its ghosting information */`
  - **if** $M_j^g \leftarrow$ *hasGhosts* **then**
    - Clear ghost information;
  - **end if**
- **end foreach**
- `/* Step 3: Delete the collected ghost entities */`
- **for** $d \leftarrow g$ *to 0* **do**
  - Remove $M_k^d \in$ *entitiesToRemove*$[d]$;
- **end for**

end

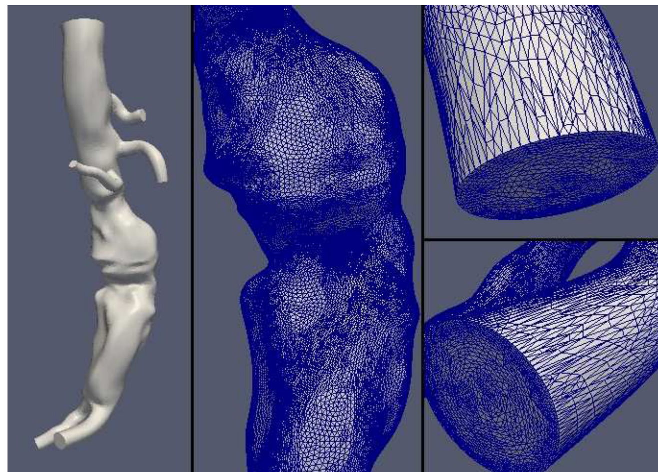


Fig. 7. Geometry and mesh of an AAA model (133 million test case) [26,36]. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130361.)

rate for each core is 8.4 GFlops/core making it 201.6 GFlops/node. By default, the compute nodes run a restricted low-overhead operating system optimized for high performance computing called "Cray Linux Environment" [14].

All available cores per node on both machines were used for the tests. For strong scaling results, the tests were executed on 512 to 32,768 processors on each supercomputer and scalability was measured using the following equation. The scalability equation consists of (1) $n_{proc\text{–}base}$: the number of base MPI processes. For our scaling study, the number of base MPI processes is 512. (2) $time_{base}$: execution time of algorithm on base MPI processes $n_{proc\text{–}base}$ (i.e. execution time on 512 processes for this study). (3) $n_{proc\text{–}test}$: the number of processes used in the test case (for our study it ranges from 1024 till 32,768). (4) $time_{test}$: the algorithm execution time on the number of processes $n_{proc\text{–}test}$. The execution time is measured in seconds.

$$scalability = (n_{proc\text{–}base} * time_{base}) / (n_{proc\text{–}test} * time_{test}) \quad (1)$$

Table 4 demonstrates the execution time of ghost creation algorithm with 1-layer on input parameters $\{g = 3, b = 0, numLayer = 1\}$ (ghost dimension: region, bridge dimension: vertex). Table 5 demonstrates the execution time of ghost deletion algorithm with 1-layer on the same input parameters. As the strong scaling tests address a fixed-size problem, with the increase in core count, a larger percentage of mesh entities are between inter-part boundaries. This leads to growing number of entities being ghosted, and more inter-processor communication is needed, whereas the parts are being less loaded with mesh entities. This explains why the scaling for ghost creation process starts falling with increasing processor count (from 1024 processors and onwards, the scaling factor decreases). There is a superlinear scaling for the first step in the AAA mesh as the entities to be ghosted decrease slightly (33.29 M to 30.6 M) when stepping from 1024 to 512 parts. This happened due to the fact that the ghosting process depends on mesh partitioning, the number of ghost entities is less if the mesh partitioning algorithm keeps maximum adjacent entities on the same part [4,25].

For the ghost deletion algorithm, once again the number of mesh entities to be deleted grow with increasing processor count which leads to reduced scalability. Only in the case of 1024 processors, when the number of ghost entities to be deleted slightly decrease from ghost entity count at 512 processors, there is a super-linear scaling.

It can be seen from Table 4 that the Cray was able to perform ghost creation faster than BG/P; one rea-

Table 4

1-layer ghost creation execution time (s) on Cray XE6 and Blue Gene/P (N/proc is number of MPI processes, E/ghosted is the number of ghost entities created)

Test case	Machine	N/proc	512	1024	2048	4096	8192	16384	32768
AAA 133M	Cray XE6	E/ghosted	33.29M	30.6M	40.19M	52.34M	68.29M	87.98M	115M
		Time (s)	30.88	11.86	7.87	5.36	3.53	2.31	1.67
		Scaling	1	1.30	0.98	0.72	0.55	0.49	0.30
	Blue Gene/P	E/ghosted	33.29M	30.6M	40.19M	52.34M	68.29M	87.98M	115M
		Time (s)	134.09	49.80	34.22	23.01	15.07	9.95	7.01
		Scaling	1	1.34	0.98	0.73	0.56	0.42	0.30

Table 5

1-layer ghost deletion execution time (s) on Cray XE6 and BG/P (E/deleted is the number of ghost entities deleted)

Test case	Machine	N/proc	512	1024	2048	4096	8192	16384	32768
AAA 133M	Cray XE6	E/deleted	33.29M	30.6M	40.19M	52.34M	68.29M	87.98M	115M
		Time (s)	1.65	0.61	0.42	0.25	0.16	0.10	0.06
		Scaling	1	1.35	0.98	0.825	0.65	0.51	0.43
	BG/P	E/deleted	33.29M	30.6M	40.19M	52.34M	68.29M	87.98M	115M
		Time (s)	4.00	1.67	1.05	0.71	0.47	0.27	0.2
		Scaling	1	1.19	0.95	0.71	0.53	0.46	0.31

son for this is the difference in processor speed (2.1 GHz vs. 850 MHz). Another factor that comes into play is the nature of the ghost creation process which is communication intensive, Cray has 24 cores/compute node (vs. 4 cores/compute node of BG/P), so much of the communication is carried out within the same node resulting in faster data exchange. As the ghost deletion process employs no communication, it can be seen from Table 5 that Cray performs approximately 2.4 times faster than BG/P which is solely due to the difference in processor speed.

Figure 8 shows how scalability varies as the number of ghost entities, being created, increase. Figure 9 compares scalability with ghost entities deleted. As the number of cores get doubled, the ghost entities to be created and deleted also increase which brings down the strong scaling factor.

To test the performance of N-layer ghost creation algorithm with different number of layers, the AAA test case was executed on 1024 processors on Cray XE6 and BG/P with number of layers from 1 to 5. Table 6 demonstrates the execution time of N-layer ghost creation algorithm in seconds with input parameters $[g = 3, b = 1, n = 1 \dots 5]$ (ghost dimension: region, bridge: edges). Table 6 shows how the number of ghosted entities increase with increasing layer count by an amount comparable to the number of entities in the first layer. As the number of layer increases to 5, the number of ghost entities reach close to the size of the mesh. The efficiency of N-layer ghost creation algorithm denoted as E_n is based on the execution time of creating 1 ghost layer on 1024 processors and the increase in total number of ghost entities after ghost layers are added, $E/ghosted$. It is calculated using

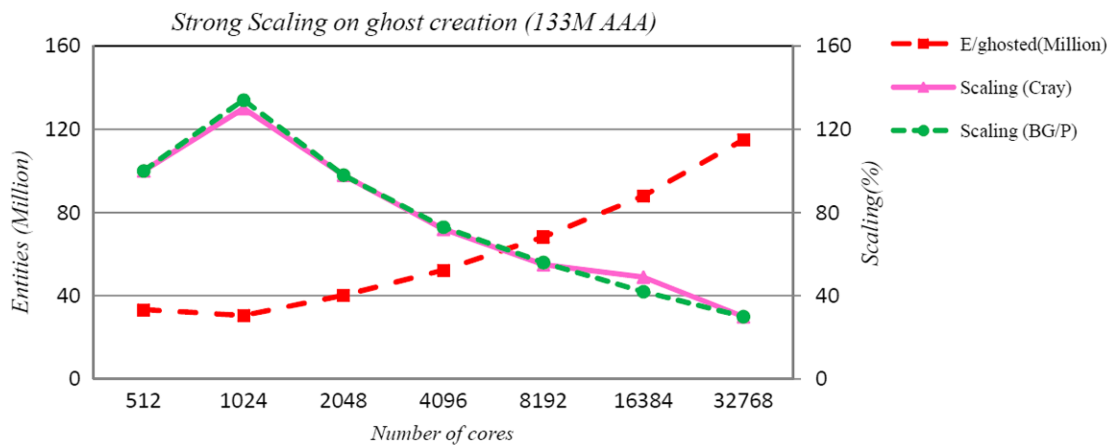


Fig. 8. An increase in Ghosted entities created per part lead to decreased ghost creation scalability on Cray XE6 and BG/P (test case: 133M air bubbles mesh). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130361.)

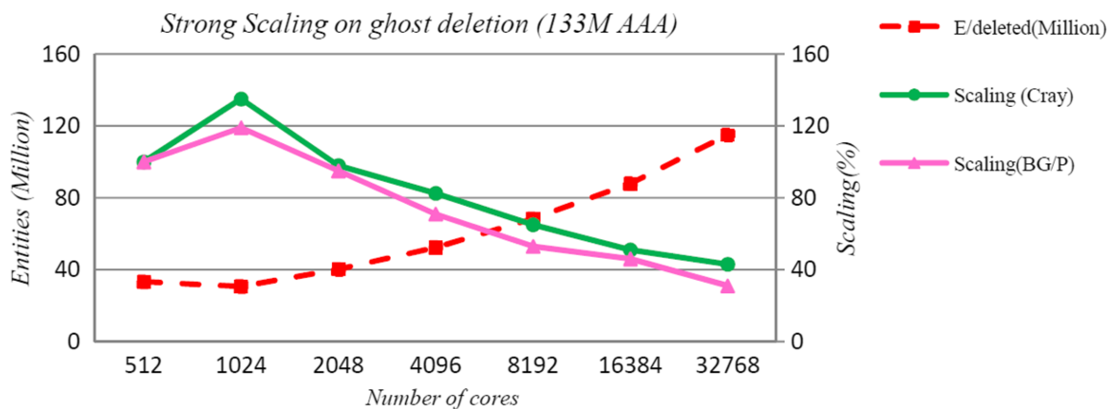


Fig. 9. An increase in ghosted entities deleted per part lead to decreased scalability on Cray XE6 and BG/P (test case: 165M air bubbles mesh). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130361.)

Table 6

N-layer ghost creation execution time (s) on 1024 processors (E/ghosted is the number of ghost entities created, E_n is the efficiency of N-layer ghost creation algorithm)

Test case	Machine		$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
AAA 133M	Cray XE6	E/ghosted	17M	37.9M	57.6M	76.5 M	93.7M
		Time (s)	8.86	16.2	21.55	26.12	30.8
		E_n	1	1.21	1.15	1.52	1.58
	BG/P	E/ghosted	17M	37.9M	57M	76M	93.7M
		Time (s)	35.44	64.80	87.1	105.67	122.70
		E_n	1	1.19	1.36	1.49	1.59

Table 7

Weak scaling of ghost creation/deletion algorithms

N/Proc	Mesh size	t_{create}(s)	t_{del}(s)	Avg. ghosts/Proc	S_G	S_{t_create}	S_{t_del}
32	265,872	0.45	0.18	2,547	1	1	1
256	2,126,662	0.49	0.19	2,889	0.88	0.92	0.95
2048	17,010,572	0.80	0.29	3,266	0.78	0.56	0.62
16384	136,084,576	0.61	0.21	3,370	0.76	0.76	0.86

Notes: t_{create} is the execution time (s) to create ghost entities, t_{del} is the execution time (s) to delete ghost entities, S_G is scaling result based on average number of ghost entities, S_{t_create} and S_{t_del} is scaling based on execution time of ghost creation and deletion algorithms.

$$E_n = \frac{(\mathit{time}_{base} * (1+\uparrow \text{ in E/ghosted}))}{\mathit{time}_{test}}. \quad (2)$$

It can be seen that as the number of ghost layers increase, the efficiency of the N-layer algorithm also keeps on increasing by this measure. This is due to the fact that as the number of ghost layers increase, the part boundary entities diminish from the ghost candidates as they are already handled in inner-most ghost layers (mostly in the 1st ghost layer) and thus one additional round of communication which is being done in Algorithm 4 solely for part boundary entities is not carried out which results in improved efficiency.

Table 7 presents the weak scaling results of ghost creation and deletion algorithm on input type [$g = 3$, $b = 0$], i.e. ghost regions, bridge vertices. The test case used for the weak scaling study is the cube test case shown in Fig. 10. The test series increases the number of processors from 32 to 16,384 in multiple of 8 on the Cray XE6 machine. The mesh size also varies in multiples of 8 (app.) along with the increasing processor count. S_G represents scaling of average number of ghost entities per processor based on 1024 processors.

$$S_G = \frac{\text{Avg. no. of ghosts on } n/\mathit{proc}_{base}}{\text{Avg. no. of ghosts on } n/\mathit{proc}_{test}} \quad (3)$$

S_{t_create} and S_{t_del} represent scalability of ghost creation and deletion algorithm based on the execution time on 1024 processors. Scalability by execution time can be calculated using Eq. (4)

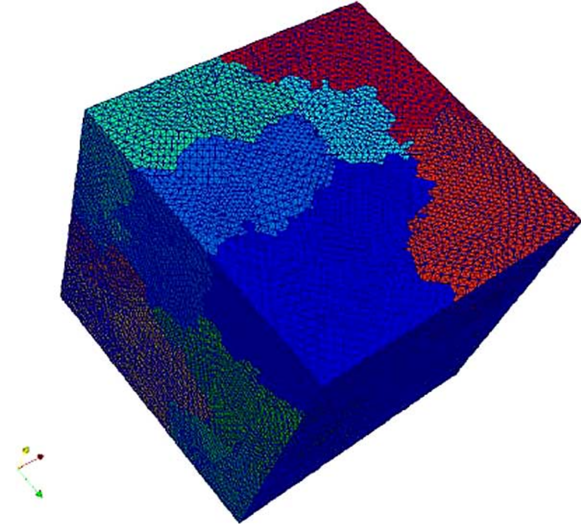


Fig. 10. The CUBE test case (adapted mesh). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130361.)

$$S_t = \mathit{time}_{base}/\mathit{time}_{test}. \quad (4)$$

The average number of ghost entities in Table 7 varies as number of ghosts are dependent on entity

neighborhood which is based on graph partitioning and can be different in every case [4,25]. When the processor count increases by a factor of 8, inter-part boundaries will also increase proportionately but the graph partitioning and load balancing algorithms will try to keep adjacent entities on the same part to reduce inter-part communication. Table 7 shows a jump of 0.31 s (resp. 0.10 s) in t_{create} (resp. t_{del}) between 256 and 2048 processors. The time spent in ghost creation/deletion is measured by taking the maximum time among all processes. As the number of ghost creation messages (resp. ghost entities to be deleted) depends on mesh partitioning and neighborhood, each part will have variable work load assigned and the part having greater ghost messages to process (resp. ghost entities to delete) will take more time which effects the scaling factor.

6. Usage example: Parallel Super-convergent Patch Recovery (SPR) based error estimator

This section demonstrates the performance results of the ghosting procedure applied to an application. A weak scaling performance study is presented to show that the ghosting algorithm, when applied to applications like SPR based error estimation, can scale efficiently on massively parallel architectures using high core count of 32,768 processors.

In a parallel adaptive finite element simulation, *a posteriori* error estimation and correction indication is an important component. A parallel error estimation procedure has been developed based on the Superconvergent Patch Recovery (SPR) scheme [37]. In the error estimation procedure, a C^0 continuous gradient field is recovered from the original C^{-1} discontinuous field, and the error is defined as the difference between the recovered and original fields.

A key step of the recovery of a given nodal degree of freedom (associated with a mesh vertex, M_i^0) employs a local least-square fitting scheme over the patch of elements (mesh regions $\{M_i^0\{M^3\}\}$) surrounding the node (or the mesh vertex M_i^0). Therefore, the complete field information of the nodal patch is required. In the context of a parallel analysis based upon a distributed mesh, a set of mesh vertices is characterized as being located on mesh partition boundaries. Consequently, the corresponding nodal patch of such a vertex is distributed among several mesh partitions, thus not complete within a local part. In order to form a complete patch on a local mesh part, and in the meantime, to avoid extensive communication cost between mesh parts, the parallel SPR based error estimator was written to take advantage of the ghosting functionality by creating a layer of ghost mesh regions.

Algorithm 7 gives the steps for the error estimation procedure using ghosting. Step 1 loads the mesh and solution data and sets up the nodal field correctly.

Algorithm 7. Error_Estimator(M, F)

Data: Distributed Mesh M, Solution field data F

Result: Calculate the error field, elemental error indicator and new desirable mesh size

```
begin
    /* Step 1: Load the mesh and solution data, setup the nodal field
       correctly                                                     */
    Mesh loading and field data setup;
    /* Step 2: Create one layer of ghost regions with bridge vertices */
    Create_Ghosts(M, REGION, VERTEX, 1);
    /* Step 3: Conduct the on-part SPR procedure                     */
    On-part (serial) SPR Procedure applied on M and F;
    /* Step 4: Calculate the error field, elemental error indicator and new
       desirable mesh size using Equations 5, 7 and 6                */
    Error field computation;
    Elemental error indication and new mesh size calculation;
    /* Step 5: Delete the ghost entities                             */
    deleteGhostEnts(M);
end
```

Step 2 creates "1" ghost layer of mesh regions on part boundaries using bridge dimension "VERTEX". The field data attached to the mesh entities is also carried along with the ghosts. After the ghosting process is done, it is guaranteed that each mesh vertex on any local part (including the ones on part boundary) has a complete local patch of elements on the same mesh part. Therefore no extra inter-part communication is required to form the local nodal patch.

In Step 3, each local mesh partition including the ghosted entities is regarded as an independent serial mesh since further communication is no longer needed. For each non-ghosted mesh vertex on the local part, the SPR procedure recovers the C^0 gradient field ε^* based on the least-square fitting scheme over the complete nodal patch [33]. Note that the ghosted mesh vertices are not processed by the SPR procedure since they are essentially duplicated copies of certain non-ghosted entities on other mesh parts.

Step 4 calculates the error field, elemental error indicator and desirable mesh size. As the exact solution ϵ is unknown, the error e_ϵ can only be estimated. In this SPR based approach, the recovered solution ε^* is used to replace the exact solution. The error field computation equation is done using the following Eq. (5) from reference [33]:

$$e_\epsilon \approx e_\varepsilon^* = \varepsilon^* - \varepsilon^h. \quad (5)$$

After the error field computation, the elemental error indication is carried out by integrating of the error over the element domain. The desirable mesh size is calculated through an h-adaptive procedure based on Eq. (6) from reference [33]:

$$h_e^{new} = h_e^{current} \times r_e, \quad (6)$$

where h_e^{new} and $h_e^{current}$ denote the new and current mesh sizes respectively. And the size scaling factor r_e is computed based on the Eq. (7) from reference [33]:

$$r_e = \|e_\varepsilon\|_e^{\frac{2}{2p+d}} \left(\frac{\eta^2 \|\varepsilon^*\|^2}{\sum_{i=1}^n \|e_\varepsilon\|_i^{\frac{2d}{2p+d}}} \right)^{\frac{1}{2p}}, \quad (7)$$

where d and p are respectively the dimension and polynomial order of elements in the part mesh. e is an element in the mesh. The goal is to ensure the relative percentage error in L_2 norm of the error in the selected quantity η is below a given limit.

Due to the ghost layers, there is no inter-part communication required in Step 4. After the error estimation procedure is completed, Step 5 deletes the ghost entities that were created as part of Step 2. With the application of ghosting, the overall communication cost of the parallel error estimator can be reduced to the one-time ghosting process in Step 2 as there is no other form of inter-part communication required.

Figure 11 gives the visualizations of the various steps of the parallel SPR error indicator. The visualizations are demonstrated on a finite element based electro-magnetic analysis example problem using ParaView [13].

A weak scaling study of the parallel SPR error estimator was carried out on Hopper, Cray XE6 from 64 to 32,768 processors. The test case used for the study is the electro-magnetic analysis example problem in Fig. 11(a). Uniform mesh refinement was carried out to generate partitioned meshes of desirable sizes (multiples of 8). Table 8 gives the weak scaling results of the error estimator where S_t represents its scalability which is calculated using Eq. (4). The execution time is the average time the error estimator procedure spends in computation (Algorithm 7 Steps 2–5). S_{avg} represents the scaling of average number of ghost entities per processor which is calculated using Eq. (3). Once again, the weak scaling factor drops slightly as the inter-part boundaries increase proportionately with increasing processor count which subsequently cause the average number of ghost entities per processor to increase.

To further demonstrate the usefulness of the ghosting algorithm applied to the parallel SPR error estimator, a similar parallel SPR algorithm without ghosting is also being developed. Algorithm 8 gives the outline of the procedure.

As given in Algorithm 8, the SPR procedure without ghosting is performed in two separate steps (Steps 2 and 4) in order to treat the interior and boundary mesh vertices accordingly whereas there is just one step of SPR in the algorithm with ghosting (Step 3 of Algorithm 7). In terms of communication costs, two rounds of inter-process communication are required to collect the input and synchronize the output data of the SPR calculation across part boundaries without ghosting (Steps 3 and 5 of Algorithm 8), which is not as efficient as the single step ghosting process (Step 2 of Algorithm 7). Therefore, it is more advantageous to apply ghosting procedure to the parallel SPR error estimator in terms of computational performance.

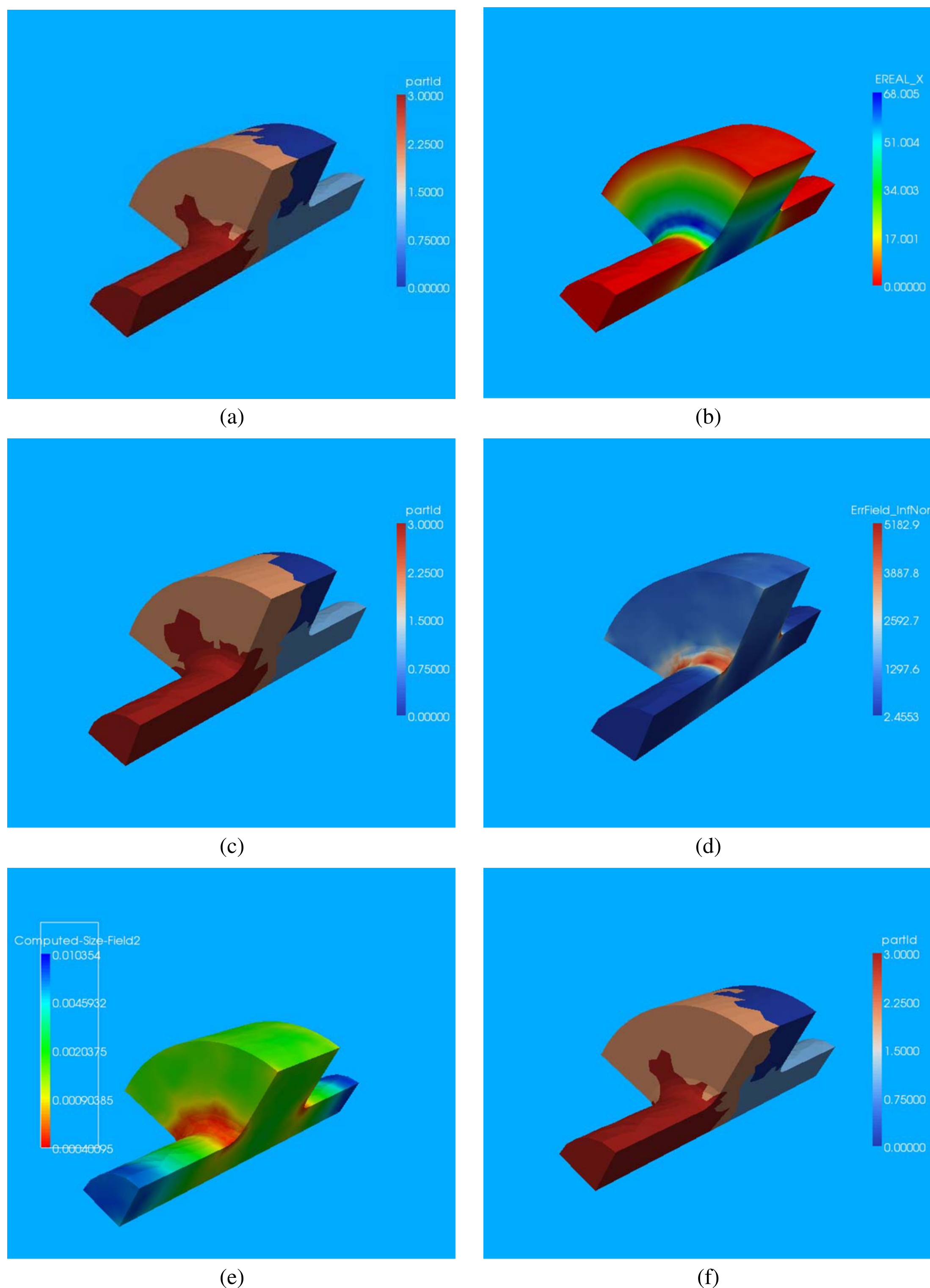


Fig. 11. SPR Error Estimation with ghosting applied on example electromagnetic mesh. (a) Initial mesh (Step 1). (b) Initial field data (Step 1). (c) Mesh after 1-layer ghosting (Step 2). (d) Error field data after on-part SPR procedure (Step 3). (e) Size field data after Step 4 (error field calculation and new desirable mesh size). (f) Mesh after Ghost deletion (Step 5). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130361.)

Table 8

Weak scaling of SPR Error Estimator Procedure (S_{time} and S_{avg} are the scaling results based on execution time and average number of ghosts resp.)

N/Proc	Mesh size	Time (s)	Avg. ghosts/Proc.	S_{time}	S_{avg}
64	61,8432	10.30	5,214	1	1
512	4,947,456	10.21	5,629	1	0.92
4,096	39,579,648	10.81	5,928	0.95	0.88
32,768	316,637,184	12.7	6,270	0.81	0.83

Algorithm 8. Error_Estimator_Without_Ghosting(M, F)

Data: Distributed Mesh M, Solution field data F

Result: Calculate the error field, elemental error indicator and new desirable mesh size

begin

```
/* Step 1: Load the mesh and solution data, setup the nodal field
   correctly */
```
Mesh loading and field data setup;
```
/* Step 2: Perform the SPR procedure to the interior mesh vertices that
   have complete on-part (local) patches */
```
On-part (serial) SPR Procedure applied to M^0s on part interior;
```
/* Step 3: Collect field data across multiple parts to form complete
   patches for part boundary mesh vertices */
```
Migrate necessary field data associated with entities on remote mesh parts to the owner part;
```
/* Step 4: Perform the SPR procedure to the part boundary mesh vertices
   after the complete patches are formed */
```
Serial SPR Procedure applied to M^0s on part boundary;
```
/* Step 5: Synchronize the recovered field data across the part
   boundaries */
```
Update the remote copies with the recovered field of the owner copies;
```
/* Step 6: Calculate the error field, elemental error indicator and new
   desirable mesh size using Equations 5, 7 and 6 */
```
Error field computation;

Elemental error indication and new mesh size calculation;

end

7. Related work

ParFUM [19,35] is a parallel mesh database that implements the classic node-element structure. Mesh parts are called chunks and the communication between chunks is done implicitly through shared and ghost entities. ParFUM allows the creation of ghost layers on the boundary of each chunk so that applications can access solution data from its neighboring elements. The ghosting process can be done through different bridging dimension (not only vertices). Multiple ghost layers can be added by calling the same ghosting routine repeatedly, which adds ghost layers one after the other [18]. However, the algorithm only creates the highest dimension ghost element and ghost nodes in a mesh. Each time new elements are added or ghost elements are deleted, a round of communication is carried out to update the ghosts or ghost owners.

The SIERRA's FEM has mesh entities on the boundary of a partition that can be shared with other parts, although only one is chosen the owner of the entity [29]. A mesh entity that resides in a process and is not in the closure of the process's owned subset is termed a *ghost* mesh entity. Multiple independent computations may require their own particular subsets of parallel ghosted mesh entities. These computation-specific ghosted subsets are mesh bulk data that is created, owned, modified and destroyed by an applica-

tion. SIERRA uses ghost entities in computations that gather field data from an owned entity's neighbors [8].

Reference [7] presents Liszt, a domain specific language for constructing mesh-based PDE solvers. As PDE solvers need to run on large distributed memory machines, the idea is to reduce synchronization between the nodes. For that, the ghost elements are created with field data and when their field values become stale, messages are exchanged between nodes to send updated values during the computational phase of the solver.

A ghost exchange method in a parallel mesh representation is presented in [31]. Ghost exchange of faces and regions are carried out for a library MOAB, used for representing, querying and modifying structured and unstructured meshes. Performance data for 32M hex and 64M tet meshes is also presented.

Reference [2] presents a finite element formulation for adaptive refinement of unstructured meshes to accurately model plastic transition zones. For parallel computation, the mesh is partitioned into sub-meshes assigned to each processor and the mesh is a global data structure such that (1) each element knows and can access the elements with which it shares an edge, (2) each vertex is able to access all the elements to which it is adjacent, in some cases these elements are owned by other processors. These two goals are accomplished by having ghost copies of the non-local elements which can also be used for higher order elements. As each ghost element should have up-to-date copies of the solution information, MPI scatter routines are used to coordinate the updates to the ghost copies and related data. The approach ghosts only top level elements in a 3D mesh.

Most of above mesh libraries [2,7,19,35] use communication to update the stale ghost copies which is done through MPI collective communication. None of the above libraries make use of message packing or neighborhood communication to avoid collective operations and communication overheads. Moreover, no scalability results or performance data of the ghost creation and deletion algorithm are provided. Our ghosting algorithm also minimizes communication costs by localizing ownership information (Section 3.2). It avoids the use of any search operations that traverse the entire mesh. The above mesh libraries only appear to support ghosting for top-level mesh entities for e.g. regions in a 3D mesh or faces in a 2D mesh, our algorithm gives flexibility to the application by creating ghosts of any topological dimension apart from the top-level mesh entities. Some of the above libraries apply their ghosting procedures to access solution field data from neighboring elements [29,35] though there is no scalability data available to analyze their performance. Some of the above libraries [2,7,19] appear to support the creation of a single ghost layer only whereas our ghosting algorithm can be used to create multiple ghost layers.

8. Closing remarks

In distributed meshes, ghosting provides an application with complete parallel neighborhood information. We have presented a N-layer ghost creation and deletion algorithm that provides ghost data for all dimensions (1D, 2D or 3D) in a distributed mesh. The algorithm supports any number of ghost layers as specified by the application. Strong and weak scaling of the ghosting algorithm on two massively parallel architectures up to a core count of 32,768 processors were presented. With a fixed problem size, as the processor count increases, inter-part communication also increases while computational work load decreases, which in turn affects the number of ghost entities. We have also presented the weak scaling results of the ghosting algorithm when applied to the parallel SPR error estimating procedure on a core count of 32,768 processors. The algorithm can further be applied to reduce the communication requirements of other applications like parallel mesh smoothing and geometric reconstruction procedures such as subdivision surfaces.The algorithm implementation is available as part of the FMDB implementation at http://redmine.scorec.rpi.edu/projects/fmdb.

Additional efforts on scalability of ghosting algorithm to take advantage of multi-threaded MPI processes are desired. As HPC systems continue to grow both in node count and core counts per node, a trend of hybrid programming model that accounts for multi-core nodes by using threads within MPI processes is emerging. The motivation is to extend the ghosting algorithm for multi-threaded MPI processes. This can potentially lead to reduced memory consumption and improved scalability. We believe that by incorporating a hybrid programming model with threads and message-passing, we can significantly reduce the communication requirements of the ghosting algorithm by utilizing the on-node shared memory for accessing non part-boundary neighborhood data. Another significant aspect in improving scalability is to make an even workload distribution during ghost creation and dele-

tion. One step in this direction is to follow a round-robin approach for distributing duplicate messages instead of the static approach using minimum part ID assignment given in Section 4.2.3.

Acknowledgement

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 Inter-operable Technologies for Advanced Peta-scale Simulations (ITAPS) program, under grant DE-FC02-06ER25769.

References

[1] Argonne National Laboratory, The Message Passing Interface (MPI) standard library, available at: http://www-unix.mcs.anl.gov/mpi, 2011.

[2] W.J. Barry, M.T. Jones and P.E. Plassmann, Parallel adaptive mesh refinement techniques for plasticity problems, *Advances in Engineering Software* **29**(3–6) (1998), 217–225.

[3] M.W. Beall and M.S. Shephard, A general topology-based mesh data structure, *International Journal for Numerical Methods in Engineering* **40**(9) (1997), 1573–1596.

[4] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R. Heaphy et al., Hypergraph-based dynamic load balancing for adaptive scientific computations, in: *2007 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2007, p. 68.

[5] Cray XE6 system, available at: http://www.cray.com/Products/XE/CrayXE6System.aspx, 2011.

[6] K. Devine, L. Diachin, J. Kraftcheck, K.E. Jansen, V. Leung, X. Luo, M. Miller, C. Ollivier-Gooch, A. Ovcharenko, O. Sahni et al., Interoperable mesh components for large-scale, distributed-memory simulations, *Journal of Physics: Conference Series* **180** (2009), 012011.

[7] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy et al., Liszt: A domain specific language for building portable mesh-based PDE solvers, 2011.

[8] H.C. Edwards, A. Williams, G.D. Sjaardema, D.G. Baur and W.K. Cochran, SIERRA Toolkit computational mesh conceptual model, *Sandia National Laboratories SAND Series, SAND2010-1192*, 2010.

[9] R. Espinha, W. Celes, N. Rodriguez and G.H. Paulino, ParTopS: compact topological framework for parallel fragmentation simulations, *Engineering with Computers* **25**(4) (2009), 345–365.

[10] I.T. Foster, *Designing and building parallel programs: concepts and tools for parallel software engineering*, Addison-Wesley, 1995.

[11] L. Freitag, M. Jones and P. Plassmann, A parallel algorithm for mesh smoothing, *SIAM Journal on Scientific Computing* **20**(6) (1999), 2023–2040.

[12] M. Gilge, IBM system blue gene solution: Blue gene/q application development, *IBM Redbook Draft SG24–7948–00*, 2012.

[13] A. Henderson, J. Ahrens and C. Law, *The ParaView Guide*, Kitware Clifton Park, NY, 2004.

[14] Hopper 2, Cray XE6 at NERSC, available at: http://newweb.nersc.gov/users/computational-systems/hopper, 2011.

[15] IBM guide to using Blue Gene/P, available at: http://www.redbooks.ibm.com/abstracts/sg247287.html.

[16] iMeshP Interface Documentation, available at: http://www.itaps.org/software/iMeshP_html/index.html, 2011.

[17] ITAPS: The Interoperable Technologies for Advanced Petascale Simulations center, available at: http://www.itaps.org, 2011.

[18] L.V. Kale, R. Haber, J. Booth, S. Thite and J. Palaniappan, An efficient parallel implementation of the spacetime discontinuous Galerkin method using charm++, in: *Proceedings of the 4th Symposium on Trends in Unstructured Mesh Generation at the 7th US National Congress on Computational Mechanics*, 2003.

[19] O.S. Lawlor, S. Chakravorty, T.L. Wilmarth, N. Choudhury, I. Dooley, G. Zheng and L.V. Kalé, Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications, *Engineering with Computers* **22**(3) (2006), 215–235.

[20] M. Mubarak, A parallel ghosting algorithm for the flexible distributed mesh database (FMDB), Masters thesis, School Rensselaer Polytechnic Institute, 2011.

[21] W. Oed, Scalable computing in a hybrid system architecture, in: *High Performance Computing on Vector Systems 2008*, Springer, 2009, pp. 13–21.

[22] 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 Transactions on Mathematical Software (TOMS)* **37**(3) (2010), 29.

[23] A. Ovcharenko, D. Ibanez, F. Delalondre, O. Sahni, K.E. Jansen, C.D. Carothers and M.S. Shephard, Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications, *Parallel Computing* (2012), to appear.

[24] A. Ovcharenko, O. Sahni, C.D. Carothers, K.E. Jansen and M.S. Shephard, Subdomain communication to increase scalability in large-scale scientific applications, in: *Proceedings of the 23rd International Conference on Supercomputing*, ACM, 2009, pp. 497–498.

[25] ParMETIS – Parallel Graph Partitioning and Fill-reducing Matrix Ordering, available at: http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview, 2011.

[26] O. Sahni, K.E. Jansen, C.A. Taylor and M.S. Shephard, Automated adaptive cardiovascular flow simulations, *Engineering with Computers* **25**(1) (2009), 25–36.

[27] E. Seegyoung Seol and M.S. Shephard, Efficient distributed mesh data structure for parallel automated adaptive analysis, *Engineering with Computers* **22**(3) (2006), 197–213.

[28] E.S. Seol, FMDB: flexible distributed mesh database for parallel automated adaptive analysis, PhD thesis, Rensselaer Polytechnic Institute, 2005.

[29] J.R. Stewart and H.C. Edwards, A framework approach for developing parallel adaptive multiphysics applications, *Finite Elements in Analysis and Design* **40**(12) (2004), 1599–1617.

[30] D. Sulsky, S.-J. Zhou and H.L. Schreyer, Application of a particle-in-cell method to solid mechanics, *Computer Physics Communications* **87**(1) (1995), 236–252.

[31] T.J. Tautges, J.A. Kraftcheck, N. Bertram, V. Sachdeva and J. Magerlein, Mesh interface resolution and ghost exchange in a parallel mesh representation, in: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, IEEE, 2012, pp. 1670–1679.

[32] J.-L. Vay, P. Colella, J.W. Kwan, P. McCorquodale, D.B. Serafini, A. Friedman, D.P. Grote, G. Westenskow, J.-C. Adam, A. Heron et al., Application of adaptive mesh refinement to particle-in-cell simulations of plasmas and beams, *Physics of Plasmas* **11** (2004), 2928.

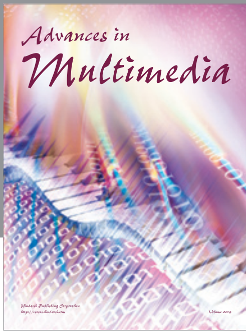
[33] J. Wan, *An automated adaptive procedure for 3D metal forming simulations*, PhD thesis, Rensselaer Polytechnic Institute, New York, 2006.

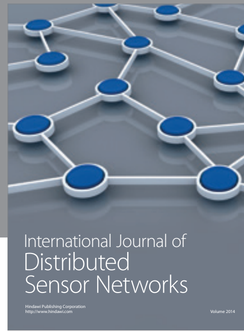
[34] J. Wan, S. Kocak, M.S. Shephard and D. Mika, Automated adaptive forming simulations, in: *Proceedings, 12th International Meshing Roundtable*, Citeseer, 2003, pp. 323–334.

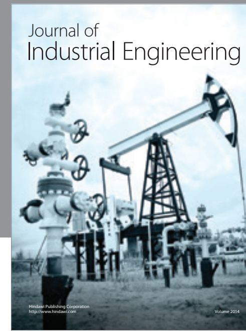
[35] X. Zeng, R. Bagrodia and M. Gerla, GloMoSim: a library for parallel simulation of large-scale wireless networks, *ACM SIGSIM Simulation Digest* **28**(1) (1998), 154–161.

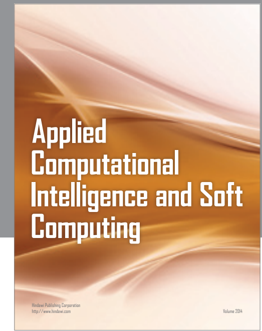
[36] M. Zhou, O. Sahni, H.J. Kim, C.A. Figueroa, C.A. Taylor, M.S. Shephard and K.E. Jansen, Cardiovascular flow simulation at extreme scale, *Computational Mechanics* **46**(1) (2010), 71–82.

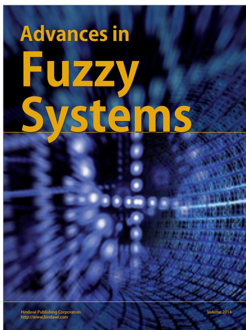
[37] O.C. Zienkiewicz and J.Z. Zhu, The superconvergent patch recovery and a posteriori error estimates. Part 1: The recovery technique, *International Journal for Numerical Methods in Engineering* **33**(7) (1992), 1331–1364.











Hindawi

Submit your manuscripts at
http://www.hindawi.com