Engineering with Computers
DOI 10.1007/s00366-009-0129-2

ORIGINAL ARTICLE

ParTopS: compact topological framework for parallel fragmentation simulations

Rodrigo Espinha · Waldemar Celes · Noemi Rodriguez · Glaucio H. Paulino

Received: 8 December 2008 / Accepted: 3 February 2009

Abstract Cohesive models are used for simulation of fracture, branching and fragmentation phenomena at various scales. Those models require high levels of mesh refinement at the crack tip region so that nonlinear behavior can be captured and physical results obtained. This imposes the use of large meshes that usually result in computational and memory costs prohibitively expensive for a single traditional workstation. If an extrinsic cohesive model is to be used, support for dynamic insertion of cohesive elements is also required. This paper proposes a topological framework for supporting parallel adaptive fragmentation simulations that provides operations for dynamic insertion of cohesive elements, in a uniform way, for both two- and three-dimensional unstructured meshes. Cohesive elements are truly represented and are treated like any other regular element. The framework is built as an extension of a compact adjacency-based serial topological data structure, which can natively handle the representation of cohesive elements. Symmetrical modifications of duplicated entities are used to reduce the communication of topological changes among mesh partitions and also to avoid the use of locks. The correctness and efficiency of the proposed framework are demonstrated by a series of arbitrary insertions of cohesive elements into some sample meshes.

Keywords Parallel fragmentation simulation · Crack branching · Cohesive elements · Topological data structure · Finite elements · Fracture

R. Espinha · W. Celes (✉)
Tecgraf, Computer Science Department, PUC-Rio,
Pontifical Catholic University of Rio de Janeiro,
Rua Marquês de São Vicente 225, Rio de Janeiro,
RJ 22453-900, Brazil
e-mail: celes@inf.puc-rio.br

R. Espinha
e-mail: rodesp@tecgraf.puc-rio.br

N. Rodriguez
Computer Science Department, PUC-Rio,
Pontifical Catholic University of Rio de Janeiro,
Rua Marquês de São Vicente 225, Rio de Janeiro,
RJ 22453-900, Brazil
e-mail: noemi@inf.puc-rio.br

G. H. Paulino
Newmark Laboratory, Department of Civil and Environmental Engineering, University of Illinois at Urbana-Champaign,
MC-250, 205 North Mathews Avenue, Urbana,
IL 61801-2397, USA
e-mail: paulino@uiuc.edu

1 Introduction

Fracture, branching, and fragmentation phenomena can be modeled by means of cohesive models of fracture. Since the cohesive model approach requires fine mesh size to capture the nonlinear behavior at the crack tip region, the associated simulations generally involve the use of highly tessellated finite element models. Currently, in order to avoid exceeding available computational resources, computational simulations are performed using geometries of reduced dimensions in comparison with original experiments. This imposes new challenges because simulations of reduced models do not fully represent original experiments due to material-dependent length scales [1]. It has also been shown that the direction of crack propagation is highly dependent on the mesh refinement level [1, 2]. As a result, accurate simulations require large amounts of computational processing power and memory space. This makes fragmentation simulations prohibitively expensive on a single traditional workstation. As a consequence, the

Published online: 27 June 2009

development of parallel environments for distributed numerical simulations has become substantially important.

On the other hand, the classical finite element mesh representation (table of nodes and element incidence) does not suffice for supporting adaptive analysis [3–5], in which the mesh geometry and/or topology may change during the course of the simulation. This is the case of dynamic fragmentation simulation based on the extrinsic cohesive element model [6–11]. Extrinsic fragmentation requires efficient identification of fractured facets (edges in 2D and faces in 3D) and for inserting cohesive elements along them. These operations change the topology of the mesh by inserting new elements and nodes, and by modifying element incidences. This cannot be performed efficiently without the support of a topological data structure. Otherwise, it would be necessary to traverse all the elements or nodes (global search) in order to gather the required information. Topological data structures usually represent some other entities, like faces, edges and vertices, rather than only the traditional elements and nodes, and need to store additional data, such as adjacency information, in order to provide efficient access to adjacency relationships among all the topological entities.

In this paper, we focus on achieving a new topological framework for supporting parallel extrinsic fragmentation simulations. Although parallel processing enables simulations to be scaled up, by decomposing computations among many processors, it also imposes some challenges that have to be addressed by system developers. These include: *dynamic mesh partitioning*, which is related to load balancing among processors; *efficient communication between partitions*; and the *adaptation of current serial algorithms for parallel execution*. In extrinsic fragmentation simulations, an appropriate parallel mesh representation that supports dynamic insertion of cohesive elements is needed in order to enable efficient execution of analysis algorithms.

We propose a topological framework for parallel representation of finite element meshes. The framework supports dynamic insertion of cohesive elements, in a uniform way, for both 2D and 3D unstructured meshes, with finite elements of any order (T3, T6, Q4, Q8, Tetr4, Tetr10, Hex8, Hex10, etc.). It is implemented as an extension to the serial topological data structure named *TopS* [4, 12, 13]. To demonstrate its capabilities, it has been implemented on top of *Charm++* [14, 15], an object-oriented framework for the development of parallel applications based on asynchronous method invocations. Of special interest, we develop a new parallel algorithm for inserting cohesive elements "on-the-fly", extending the serial algorithm proposed by Paulino et al. [13]. The parallel algorithm uses "symmetrical" modifications of duplicated entities to reduce the communication of topological changes. Yet, no access locks are required when entities are simultaneously modified in different mesh partitions. Linear scaling with the number of cohesive elements inserted is expected. Computational experiments are realized decoupled from a mechanical analysis computer code in order to test and explore the proposed topological framework.

This paper is organized as follows. Section 2 reviews some issues with distributed mesh representation and previous related research on parallel topological data structures. Section 3 presents a short review of TopS, the serial topological data structure that is extended for parallel environments. Section 4 briefly reviews the topological operations proposed in Ref. [13] for inserting cohesive elements. Section 5 describes the proposed framework for topological representation of finite element meshes in parallel. In Sect. 6, we present an algorithm for parallel insertion of cohesive elements, based on the topological framework described in Sect. 5. Next, Sect. 7 discusses some computational experiments used to demonstrate this work. Concluding remarks and directions for future work are presented in Sect. 8.

2 Motivation and related work

Distributed mesh representations [16–18] are used to provide the necessary support for finite element analyses in parallel. The approach consists of decomposing the domain into a number of partitions, or sub-domains. These partitions represent units of execution that are assigned to a group of processors. Parallel analysis provides a workaround for memory and processing bottlenecks of traditional workstations and thus permits both enlarging the model and reducing the simulation time. However, it also introduces some issues that must be addressed in order to be advantageous over the traditional serial approach, as parallel algorithms tend to be slower than serial when executed on a single processor.

One of these issues is related to communication and synchronization among partitions. For an explicit simulation model, partitions must communicate with each other in order to exchange data needed for the simulation at each time step, and the data must be available for computation in a consistent and up-to-date way. In shared memory systems [19], data communication is straightforward, as all the processors reside in the same (virtual) memory space, and thus memory can be directly used to send or receive data asynchronously. Synchronization, when needed, can be efficiently performed by the use of memory locks. On the other hand, in distributed systems [19], processors have their own private memory space and are connected by a network. In this case, message passing is the most common way to communicate data among two or more partitions, and synchronization can be done explicitly by means of specific function calls. Shared memory architecture has

been used in the development of high-performance systems, and algorithms may be simpler than for distributed memory. As a consequence, many algorithms have been developed targeting at this architecture. In the context of topological data structures, Waltz [20] has presented a set of parallel algorithms for accessing topological entity relationships useful for finite element solvers. However, distributed memory systems tend to scale better, and in the last years new massively parallel systems [21] have been developed based on a large number of computing nodes connected by a very fast network. In this paper, we focus on algorithms suitable for distributed memory systems.

Most of the time, communication occurs only among adjacent partitions. Even so, network communication overheads tend to be considerable in comparison with the serial processing at each partition. Hence, it is crucial to minimize these communications, in reference to both number and length of messages sent and received. This must be taken under consideration by applications responsible for generating and partitioning the mesh. For each element or node, the analysis code usually needs to access adjacent entities in order to compute corresponding results for the current simulation time step. Then, it is important to cluster elements or nodes in a way to minimize the number of adjacent entities belonging to different partitions, thus reducing the need for communication. Two well-known sets of serial and parallel graph partitioners are *METIS* [22, 23] and *ParMETIS* [24], the parallel version of METIS. They are able to partition and rearrange large finite element meshes, by modeling them as graphs, and can also be used in the decomposition of sparse matrices. Both METIS and ParMETIS try to distribute finite elements evenly among the partitions and to minimize the interfaces shared by different partitions.

Balancing the load among processors [25] is another important issue of parallel systems. Poorly distributed workload may eliminate all the advantages of using parallel algorithms, as the required computation time is at least the time of the busiest processor. For a simulation involving static meshes, a partitioning program like METIS can be used once in a preprocessor step to generate or repartition the distributed mesh. No dynamic load balancing scheme is needed after that, and usually one partition is assigned per processor. However, in adaptive analysis, the size of each partition (number of elements and nodes) may vary significantly after the execution of each simulation step. In this case, load balancing algorithms are important so that parallel analysis can perform efficiently. The *Zoltan* library [26] provides many utilities that may be helpful for the management of distributed dynamic meshes, including partitioning, load balancing, and communication procedures. The *Charm++* framework [14, 15] provides support for automatic load balancing in the development of parallel applications.

Efficient parallel mesh modification operations must be provided by a topological framework for dynamic meshes. These operations are used in mesh adaptation procedures and for the migration of elements between partitions during load balancing. Maintaining data structure consistency after parallel modifications may be difficult, but is a requisite for any framework that provides support for parallel adaptive analysis.

Some parallel topological data structures with support for dynamic meshes have been proposed in several papers [16–18, 27–30]. They either address specifically issues related to mesh representation or attempt to manage the entire analysis process.

2.1 MDB/PMDB [3, 27, 28]

The *Parallel Mesh Database* (*PMDB*) [27, 28] is a framework implemented on top of *Mesh Database* (*MDB*) serial data structure [3] and provides operators for manipulation of a general distributed mesh. Entities such as region, face, edge, and vertex are represented. Each region is assigned to a unique processor, while faces, edges and vertices are duplicated on the processors that contain regions incident to them. Each duplicated entity maintains a list of references to its copies in the other partitions. The framework PMDB employs the concept that an entity is owned by a single partition. The owner partition may be determined by the minimum of the tuple (*pi*, *ei*) among the list of uses of an entity, where *pi* is a processor *id* and *ei* is the entity's local *id* on that processor. Manipulation operations include querying of adjacency information and insertion and removal of entities. Mesh partitioning and load balancing procedures are also provided.

2.2 AOMD/PAOMD [17, 31]

The *Parallel Algorithm Oriented Mesh Database* (*PAOMD*) [17] extends the *Algorithm Oriented Mesh Database* (*AOMD*) [31] to support distributed meshes. It provides a general parallel mesh management framework in which mesh representation can be adapted to different types of applications. All the possible sets of entities (vertices, edges, faces and regions) and adjacencies among them can be represented; the application may select which ones are needed. Except for the vertex, an entity is represented and described by a set of entities of lower dimension and their associated ordering. Like PMDB, each partition is assigned to a processor, and the local mesh is represented by a serial AOMD mesh. Mesh entities that are classified on partition boundaries must exist in the parallel data structure and may be shared with other partitions. Connection of an entity with its representations in other partitions is made by a message that must be sent to every partition whenever the

mesh is modified. This message contains the local address of the entity and the list of the *ids* of its vertices. Moreover, PAOMD requires that each vertex must be assigned a unique global *id,* and an entity may be identified by its list of vertices. Mesh adaptivity, dynamic load balancing, and entity migration procedures are provided by PAOMD.

2.3 FMDB [18]

Seol and Shephard [18] have presented a parallel mesh infrastructure called *Flexible distributed Mesh DataBase* (*FMDB*), for general non-manifold models. Similarly to PAOMD, this framework allows each application to select the entity types that are needed to be present in the mesh. Each partition is represented by a serial mesh, with a different treatment for entities on its boundaries. Boundary entities are duplicated in all the partitions in which they are needed by adjacency relations. However, a single partition is assigned as the owner of an entity. This is the one with the least number of partition objects among the partitions in which the entity is duplicated, in order to keep load balancing during mesh modification. An algorithm for efficient migration of mesh entities based on FMDB is also presented in Ref. [18].

2.4 LibMesh [29]

The parallel framework *LibMesh* [29] provides physics-independent aspects of FEM analysis. It includes a distributed mesh data structure, adaptivity routines and interfaces to other existing libraries used by FEM applications. The data structure is based on the classic representation of elements and nodes. Each element or node is assigned a unique global *id*. In addition to its own *id*, elements also contain a processor *id*, pointers to their incident nodes (nodal connectivity), and to other adjacent elements (face neighbors). Unfortunately, despite the logical domain decomposition used to assign elements to individual processors, a complete copy of the mesh is stored on each processor. This limits the use of the framework for large-scale applications. Nevertheless, according to the authors, a fully parallelized implementation of the data structure is under consideration [29]. The libMesh framework has been explored by Wang [32, 33] in large scale topology optimization problems.

2.5 SIERRA [30]

The *SIERRA* framework [30] provides a set of general tools for supporting the development of mechanics applications. Among its features, it includes a distributed unstructured mesh data structure, along with adaptivity and load balancing, an interface to linear solvers, and support for creating multiphysics applications. The SIERRA's FEM data structure represents node, edge, face, and element entities (or objects). Edges, faces, and elements are specified by a set of vertex nodes and can be connected to other entities of different types. The connections used can be configured and additional entities or relations can also be created by SIERRA (for instance, the faces of all elements or the *ghost* elements on partition boundaries). Like other previous data structures, mesh entities on the boundary of a partition may be shared with other partitions, although only one is chosen as the owner of an entity. Entities are uniquely identified in the entire mesh by the tuple (*type*, *id*), where *type* is the type of the entity (e.g. node, element, etc.) and *id* is a unique integer among all the entities of the given type.

2.6 ParFUM [16, 34]

The framework *ParFUM* [16, 34] deals with unstructured meshes. It is based on Charm++ [14, 15], a framework for the development of parallel object-oriented applications, and *AMPI* [35], an MPI [36] implementation built on top of *Charm++*.

In fact, ParFUM implements the usual concepts of element and node, with domain attributes associated to these mesh entities. It also includes some adjacency information, like node-to-node, node-to-element, and element-to-element, useful for some types of analysis. Mesh partitions are called *chunks*, and are usually associated to exactly one MPI process [16].

Communication between chunks is implicitly done through special *shared* and *ghost* entities [16] (see Fig. 1). While elements are assigned to exactly one chunk, nodes can be shared by elements of different chunks, and thus are duplicated in each of them. These nodes are referred as *shared nodes*. Yet, during a simulation step, entities may need information from neighboring entities from other chunks. For that, ParFUM allows the creation of ghost layers on the boundary of each chunk. These layers consist

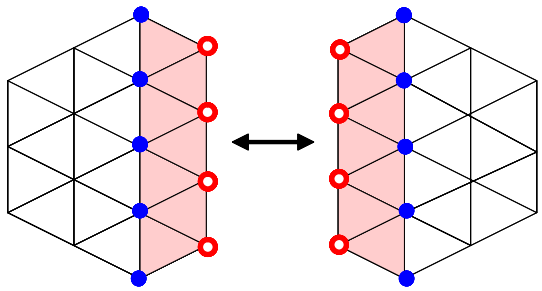


Fig. 1 In ParFUM, meshes are decomposed into distinct *chunks*. Two neighboring chunks are shown. Shared nodes are represented by *solid circles*. Those nodes lie on the interface between the chunks. A ghost layer composed of read-only copies of elements (*shaded triangles*) and nodes (*hollow circles*) from the neighboring chunk has been added around the shared nodes of each chunk

of read-only copies of neighboring entities from other partitions, referred as *ghost elements* and *nodes.* A local index is assigned to each entity in a chunk. If two chunks share nodes, in order to map between local and remote indices, both maintain a list of each other, with the local indices of the shared nodes ordered in a consistent way. Similarly, in the case of ghost entities, the chunk that has the real entity maintains a *sendghost* list, while the one with the ghost entity maintains a *receivenode* list. Collective communication procedures are provided to synchronize data stored at the entities of the ghost layer.

Notice that ParFUM supports two-dimensional incremental parallel mesh modification based on atomic operations [37]. Atomicity is ensured by the locking of entities. In this manner, if an element on the boundary of a chunk is to be modified, locks that define exclusive access to the concerning nodes are requested. When all the locks have been acquired, the modification operation takes place locally and communication messages are exchanged in order to synchronize the neighboring chunks. Finally, the acquired locks are released and can thus be requested by other interested chunks. This scheme allows adaptive algorithms to use the operators as simple serial procedures and to perform mesh modifications without the need for explicit synchronization. On the other hand, the overhead of entity locking and the number of exchanged messages can be significant. Choudhury [37] removes the overhead of locking by running multiple chunks on each physical processor in order to increase concurrency and reduce idle times.

Fracture simulations [34] are limited to 2D triangular meshes and use pre-inserted cohesive elements that are activated on demand. A complete and general topological support for parallel insertion of explicit cohesive elements, both for 2D and 3D meshes, is needed. This is the focus of the present paper.

3 TopS: serial topological data structure

A compact adjacency-based topological data structure with support for dynamic insertion of cohesive elements has been presented in Refs. [4, 12, 13] and named *TopS*. One of the benefits of TopS is that it requires small storage space while access to all adjacency relationships is provided in time proportional to the number of retrieved entities. The main concepts of TopS that are closely related to this work are briefly presented (see Refs. [4, 12, 13] for a detailed exposition of TopS).

3.1 Topological entities

The topological data structure TopS is able to represent meshes consisting of any type of elements defined by templates of ordered nodes, in 2D or 3D, under the same topological framework. Although several types of entities are defined by the data structure, only two of them are explicitly represented: *element* and *node*. As explicit entities, *node* and *element* are allocated and exist in the memory space of the data structure. *Element* represents finite elements, and has references to its boundary nodes and to its adjacent elements (i.e. elements that share a facet). *Node* represents finite element nodes, either corner or mid-side nodes, and stores its coordinates in the geometric domain and a reference to one of its incident elements. Some of the types of elements supported by TopS are shown in Fig. 2.

Other entities are implicitly represented by TopS: *vertex*, *edge* and *facet*. *Vertex* represents a corner node and may be shared by several elements. An *edge* is a one-dimensional entity bounded by two vertices and may also be shared by several elements. Edges can also contain one or more mid-side nodes (higher order). *Facet* represents the interface between two elements or between one element and the boundary of the model. For 3D models, a facet is a two-dimensional entity, which is bounded by a set of edges, while, for 2D models, it is a one-dimensional entity and corresponds to a single edge. Facets offer a convenient abstraction for implementing operations that act on the interface between elements in a uniform way for both 2D and 3D models.

The topological data structure TopS also defines additional implicit entities associated to the use of vertices, edges and facets by the elements of the mesh; these additional entities are *vertex-use*, *edge-use* and *facet-use*, respectively. The uses of a facet (facet-uses) by the two elements that share the facet are illustrated in Fig. 3. Each element in isolation is bounded by a set of local facets, edges, and vertices. These local entities are mapped to the corresponding entity-use of an element. The topology of the element in isolation depends only on element type (e.g. T3, T6, TET4, TET10). Thus, it can be used to define a fixed *template* [3, 4] that is reused by every element of the same type. This template defines a local order in which

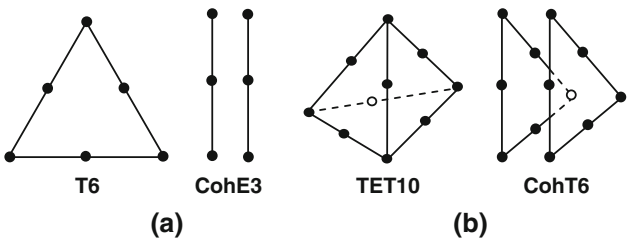


Fig. 2 Representative elements available in TopS [13]: **a** a quadratic triangle (T6) and the corresponding cohesive element (CohE3); **b** a quadratic tetrahedron (TET10) and the corresponding cohesive element (CohT6)

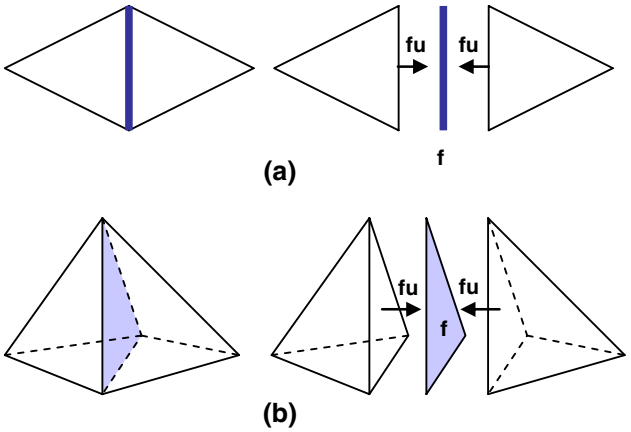


Fig. 3 A facet *f* used by the two adjacent elements in 2D **a** and 3D **b** cases. The implicit representation of the facet is given by one of the incident facet-uses

entities are referenced by the element, and provides direct access to adjacency relationships among local entities within the element.

In contrast to *element* and *node*, representations of implicit entities are created on-the-fly when requested by the application. *Facet-uses*, *edge-uses* and *vertex-uses* are each defined by the tuple (*Ei*, *lid*), in which *Ei* is the reference to one element that uses the corresponding entity and *lid* is the local identification of the entity within the element. *Facets*, *edges* and *vertices* are defined by one of their uses. In that manner, edges consisting of the same nodes can be identified as two different edges, what would not be possible if they were defined by the set of bounding nodes. It is important to note that different edges with the same bounding nodes are common in fractured models. The element of the entity-use used to represent a given facet, edge, or vertex holds an "*anchor*". The anchor is important to enable the enumeration of entities without duplications.

All the 25 possible adjacency relationships defined between every pair of entity types can be obtained in time proportional to the number of retrieved entities in TopS. The retrieval of the uses of an edge, for instance, can be done as follows: starting from the edge, we first access one of its uses directly from its representation. Then, we access the adjacent facet-uses in the element of the edge-use, by using the element template. From the facet-uses, we have access to the adjacent elements and thus to the corresponding edge-uses. This is repeated until all the uses of the edge are visited [4].

When a reference to an entity (either explicit or implicit) is requested by the application, an opaque handle is returned. An entity handle can be respectively: an element *id* (*Ei*), a node *id* (*Ni*) or the representation of an implicit entity (*Ei*, *lid*), and uniquely identifies the corresponding entity. Handles establish a uniform way in which entities are accessed in TopS. Every entity and its analysis attributes are always accessed through the corresponding handle. Thus, there is no distinction from the application point of view between either explicit or implicit entities.

3.2 Cohesive elements

The topological data structure TopS represents cohesive elements *explicitly* and treats them like any other type of element. They are defined by element templates and can hold analysis attributes. In this manner, the topology of cohesive elements can be naturally handled by TopS when the mesh is modified. This is different from other approaches, in which cohesive elements are treated as attributes attached to facets of bulk elements [8, 9].

Cohesive elements are represented by two facets (see Fig. 2). The region in between the facets represents the boundary of the mesh. Differently from bulk elements, incident nodes of a cohesive element may be duplicated, indicating that the two facets share a common node. The following incidence is thus valid for a CohE2 element (the two-dimensional linear cohesive element): $node_A$, $node_B$, $node_A$, $node_C$. Even though the same node can be referenced by the two facets, two distinct local vertex-uses are defined. These vertex-uses are each incident to exactly one facet. The same applies for shared edges in 3D cohesive elements.

4 On serial insertion of cohesive elements

In actual extrinsic fragmentation simulation, cohesive elements are inserted on demand at facets that are determined by the analysis application, according to the fracture criterion in use. When a new cohesive element is inserted, the topology of surrounding entities is modified. This requires the data structure to be correctly updated. Paulino et al. [13] have presented a systematic topological classification of fractured facets that can be applied in order to identify the topological operations needed for updating the data structure. One of the advantages of this classification is that it can be uniformly applied to any type of element, both 2D and 3D. It is briefly reviewed in this section.

Consider a cohesive element that is to be inserted at a facet shared between elements *E1* and *E2*. This facet has two associated facet-uses, *fu1* and *fu2*, respectively. The steps for updating the data structure for inserting a new cohesive element are:

1. The cohesive element is created and inserted in-between *E1* and *E2* (see Fig. 4a). The adjacency of both elements is then updated so that *E1* is no longer adjacent to *E2*, and vice versa. Now, both are adjacent to the new cohesive element.

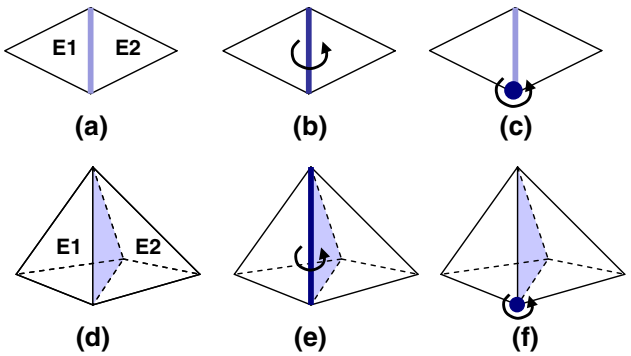


Fig. 4 Operations for updating the topology of the mesh when a new cohesive element is inserted, both in 2D (**a**–**c**) and 3D (**d**–**f**) cases. **a**, **d** A new cohesive element is inserted at the facet shared by *E1* and *E2*. **b, e** For each edge, all the corresponding edge-uses are visited in order to determine whether *E2* can be reached starting from *E1*. If *E2* was not visited, then the edge is split and the respective mid-side nodes (if they exist) are duplicated. **c**, **f** For each vertex, all the corresponding vertex-uses are visited in order to determine whether *E2* can be reached starting from *E1*. If *E2* was not visited, then the vertex is split and the respective corner node is duplicated

2. For each edge-use (*eu*) of the facet-use (*fu1*) associated to *E1*, retrieve all the other uses of the same edge considering the new adjacencies of *E1* and *E2* (see Fig. 4b). If the corresponding edge-use at the facet-use (*fu2*) of *E2* cannot be reached, the edge must be duplicated. In this case, the mid-side nodes are also duplicated, if they exist.
3. A similar procedure is done for each vertex-use (*vu*) of the facet-use *fu1* (see Fig. 4c). If the corresponding vertex-use at the facet-use (*fu2*) of *E2* cannot be reached, the vertex must be duplicated. In this case, the corresponding corner node is also duplicated.

Element connectivity must be updated for each node that is duplicated. This is done by replacing the original node for the new one in all the elements that are visited when edge-uses and vertex-uses are retrieved in steps 2 and 3. The elements that are not reached are not changed.

5 ParTopS: parallel TopS

In order to address parallel insertion of cohesive elements, we have extended the data structure named TopS [4, 12, 13] for supporting distributed meshes, creating the topological framework named ParTopS. According to the original TopS philosophy, we aim to keep ParTopS simple and compact, including the set of features that are needed for parallel insertion of cohesive elements.

Like other distributed mesh representations, the FEM model is decomposed into a set of disjoint partitions, defined by distinct subsets of the elements of the mesh. Each partition consists of a serial TopS mesh along with additional communication infrastructure. The extensions to TopS that make up ParTopS are described in the following subsections.

5.1 Communication layer

During the analysis, computations performed on elements or nodes may need to access data from adjacent entities. However, elements or nodes on the boundary of a mesh partition may be adjacent to entities that are represented in another partition. In order to reduce the amount of inter-partition communication needed to access the required data, a *communication layer* is constructed around the boundaries of each partition.

The communication layer consists of local copies of remote entities. It is built when the original mesh is first partitioned and maintained during partition's lifetime. Local copies of remote entities are represented by two different entity types: *proxies* and *ghosts*.

- *Proxy entities*, nodes and elements, are exact local copies of real entities from other partitions (remote entities). They are treated like any other local entity, and thus can be accessed, edited, and assigned attributes. Implicit entities are identified by one of their uses by surrounding elements (see Sect. 3). As a consequence, they are considered as proxies when associated to a proxy element.
- *Ghost entities* are read-only copies of remote entities. They define the boundaries of the *communication layer* (see next subsection), and thus can only be incident to proxy or other ghost entities. The main purposes of ghosts are to provide attributes that may be needed to the analysis program for computations on the incident proxy entities and to ensure local mesh consistency on the boundaries of the communication layer. As a consequence, topology of ghost entities is required to be consistent with respect to local partition, but not with corresponding entities in other partitions. For example, every node in TopS holds a reference to one of its incident elements. If we consider a ghost of the local partition, it may reference an element that is different from the one referenced by a corresponding real or proxy node in another partition. In ParTopS, any entity, except for elements, can be represented as a ghost in a given partition. Implicit entities are considered as ghosts if all of their nodes are ghosts.

Each element is naturally assigned to a single partition, while other entities, like nodes, may be used by two or more elements of different partitions (see Fig. 5). Nevertheless, in ParTopS, exactly one partition is labeled as the *owner* of any entity. This partition is responsible for

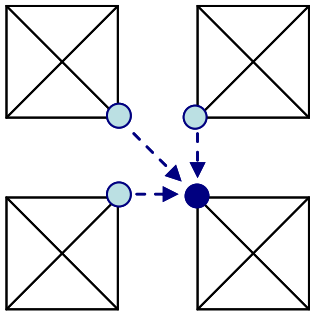


Fig. 5 A node may be shared by several partitions. However, it is owned by only one of them. In the owner partition, the node is represented as a local entity, while in the other partitions it is represented by proxy entities

representing and managing the real entity and associated attributes, while other partitions represent it by a proxy or a ghost entity.

Each proxy or ghost holds a reference to the corresponding real entity. This reference is defined by the tuple: (*owner_part, owner_handle*). Thus, *owner_part* is the index of the partition in which the real entity is represented and *owner_handle* is the serial TopS handle of the entity with respect to that partition. In ParTopS, every entity can be uniquely identified by this tuple.

In Fig. 6a, a sample mesh is decomposed into two partitions, with a communication layer added to the boundary of each one. References from a proxy and a ghost node to the corresponding real nodes are illustrated in Fig. 6b. The elements referenced by the nodes are also indicated in the figure by markers placed at them in each partition. As in serial TopS, these markers (or "*anchors*") define the implicit vertices associated to the nodes. Note that the highlighted proxy node is consistent between both partitions (global topological consistency), while the ghost node is only consistent within each partition (local topological consistency).

5.2 Construction of the communication layer

When a new mesh partition is created, some nodes and elements are first assigned to it (see Fig. 7a, b). All these entities are represented as local entities. However, elements are defined by a set of nodes, some of which may be owned by a different partition (see Fig. 7b). In order to minimize communication and to provide transparent access to remote entities, proxy nodes are created to represent the corresponding real entities in local partition. Similarly, proxy elements are inserted around all the nodes that are on the boundaries of the partition interfacing with other partitions in order to provide local access to adjacent data (see Fig. 7c). Implicit entities are represented by the elements at which they are anchored and thus are kept consistent when the corresponding local or proxy elements are constructed. The boundaries of the communication layer are treated differently. Rather than proxies, ghost nodes are created (see Fig. 7d). These entities delimit the communication layer and ensure local topological consistency of the partition.

A partition must be aware of its neighbors in order to be able to communicate efficiently. In ParTopS, two partitions are neighbors if one of them has a proxy for an entity that is owned by the other partition. Ghost entities do not influence the neighborhood of partitions, as their data can be retrieved and accessed through incident proxy elements. Partition neighborhood is determined during the construction of the distributed mesh. The algorithm for inserting cohesive elements proposed in Sect. 6 does not change the sets of neighboring partitions; the new elements and nodes, and consequently implicit entities, are assigned to partitions that are already neighbors.

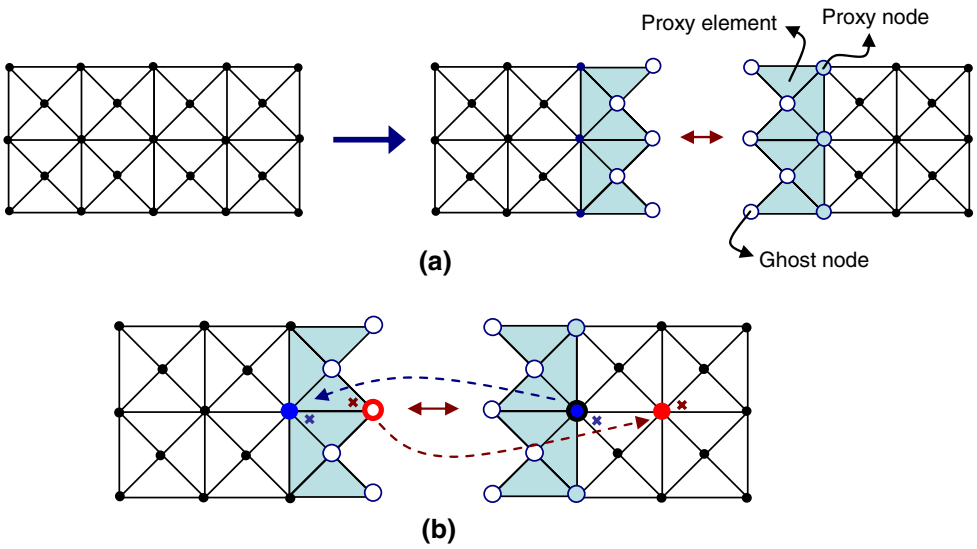


Fig. 6 **a** A mesh is decomposed into two distinct partitions. Proxy and ghost entities are added to each partition in order to represent remote entities locally. **b** A proxy and a ghost node are emphasized. The proxy node has a reference to the corresponding real entity in the other partition. The associated vertex has the same representation in both partitions (expressed by the anchor—"*x*"—placed at the same incident element). The ghost node also has a reference to the corresponding real entity. However, the representation of the associated vertex may be different at each partition

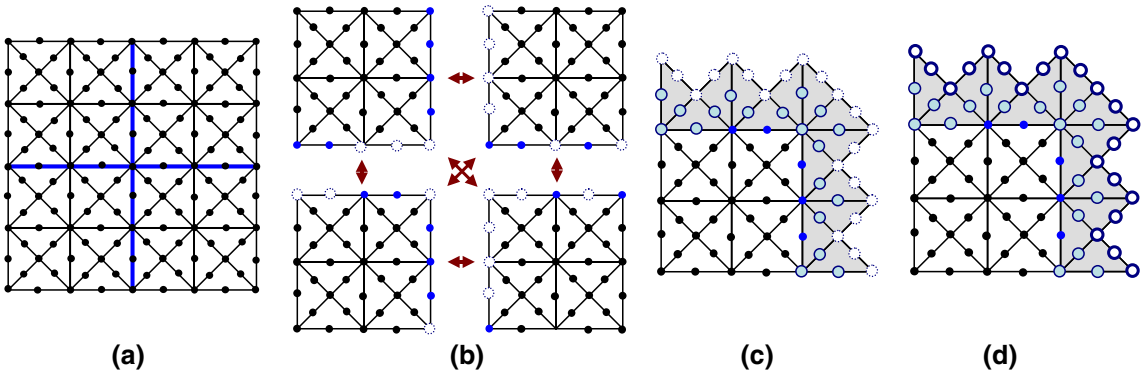


Fig. 7 Construction of the communication layer. **a** Original mesh. **b** The mesh is decomposed into four partitions. **c** One of the partitions is shown. Proxy nodes are created to represent remote nodes incident to local elements. Then, proxy elements and nodes are inserted around the boundaries of the partition. **d** Ghost nodes are inserted to fill the boundaries of the communication layer

At each simulation step, computation can be performed on local and proxy entities of each partition. Then an update procedure synchronizes the attributes of ghost entities with the corresponding real entities so that they can be accessed seamlessly by the analysis application. In order to update ghosts, a partition must request data from its neighbors. However, as ghosts are not used to determine partition neighborhood, they may refer to real entities from a partition that is not in the neighborhood set of the current one. Thus, requests for a ghost entity are directed to one of the neighboring partitions that own an element incident to it. The partition then replies with the attributes associated to the ghost entity. If we consider that topological modifications to proxy entities are symmetrical among all involved partitions, as in the algorithm of Sect. 6, then no topological synchronization is required for proxy entities, except for the newly created ones.

5.3 Implementation

The ParTopS algorithm is general and can be implemented directly in parallel environments such as MPI [36] or *Charm++* [14, 15]. To demonstrate the flexibility to interface with an existing framework, the ParTopS communication mechanism has been implemented on top of Charm++.

The software Charm++ is an efficient C++ framework for developing objected-oriented parallel applications based on asynchronous method invocations. By sending messages asynchronously, an application is able to overlap computation and communication steps. A Charm++ program is composed of parallel objects, named "*chares*", each one being assigned to a virtual processor. In fact, "chares" are automatically mapped by Charm++ to the available physical processors and may be migrated when necessary. In ParTopS, each mesh partition is implemented as an individual "chare". Moreover, Charm++ has built-in support for dynamic load balancing, which can be extended to incorporate new algorithms. Although load balancing issues were not explored in this paper, good load balancing is of great interest for adaptive analysis.

6 Parallel insertion of cohesive elements

In this section, we present a parallel algorithm for insertion of cohesive elements. Unlike previous approaches, this algorithm can be applied in a uniform fashion for 2D or 3D meshes composed of any type of element. Cohesive elements are explicitly represented, and may be used by an application like any other regular elements. They can be inserted on demand and hold analysis attributes.

For this purpose, the parallel mesh representation (ParTopS) described in the last section is combined with the systematic topological classification of facets proposed by Paulino et al. [13]. In addition, we explore the use of procedures that generate the same topological results in every partition in order to reduce the amount of inter-partition communication needed to update mesh topology.

6.1 Lock-free approach

The parallel insertion of cohesive elements is based on a lock-free approach. Therefore, no access locks are required when an entity is concurrently modified by different partitions. Although entity locks may represent a convenient way to implement other parallel mesh adaptivity operators, we have not found a simple and efficient algorithm for inserting cohesive elements based on them. The main issue regards the duplication of nodes on the boundary of a mesh's partition or the communication layer, resulting from the insertion of a new cohesive element. In order to

determine whether the nodes must be duplicated and thus keep mesh topology consistent, one partition may depend on modifications that occur in other partitions, thereby creating a cyclic dependence, which has to be addressed.

The approach adopted in this work consists in employing symmetrical topological operations on corresponding real and proxy entities and treating boundary (ghost) entities separately. As a result, our approach avoids the use of locks, thus eliminating idle times in the algorithm. In the end, the use of symmetrical operations also reduced the communication among partitions. There is still an important advantage of using a lock-free approach: it fits nicely with the asynchronous programming paradigm of the Charm++ framework.

6.2 Algorithm overview

The first step for adaptively inserting cohesive elements in the mesh is the identification of fractured facets. This has to be accomplished by the analysis application. Once fractured facets are identified, the parallel algorithm for inserting cohesive elements is performed in three phases:

Phase 1. Cohesive elements are inserted at both local and proxy fractured facets, using a serial algorithm that produces symmetrical topological results in every partition for a given facet.

Phase 2. The newly created proxy entities (cohesive elements and nodes) are updated.

Phase 3. The ghost entities affected by new cohesive elements are updated.

A sample mesh decomposed into four partitions is used in this section to illustrate the phases of the algorithm. This mesh is presented in Fig. 8. The procedure to identify fractured facets and the phases of the proposed algorithm are described in details in the next subsections.

6.3 Identification of fractured facets

At each simulation step, the analysis application identifies new fractured facets, along which cohesive elements have to be inserted (see Fig. 9). The fracture criterion can be applied to local facets only and then synchronized with the corresponding proxy facets in other partitions. So, the procedure has to be subdivided in two phases. First, at each partition, the application checks the fracture criterion on all local facets. Then, a network communication phase is needed in order to unambiguously assign the fractured classification to the corresponding proxy facets. There is no need to assign fractured classification to ghost facets because they are read-only entities and, thus, no cohesive elements are inserted along them. In Fig. 9, the sample mesh with some random fractured facets is shown. Once the update phase is concluded, the three phases of the parallel algorithm for inserting the corresponding cohesive elements are executed.

6.4 Serial insertion of cohesive elements (Phase 1)

In *Phase 1*, cohesive elements are locally inserted at fractured facets of distinct mesh partitions using the serial algorithm proposed by Paulino et al. [13] (see Sect. 4). When a cohesive element is created, some nodes may have to be duplicated. If the same topological results can be obtained for a set of entities in every partition, regardless of the order in which cohesive elements are inserted, then the topology of local and proxy entities is consistent after *Phase 1,* without the need for inter-partition communication. Ghost nodes, however, are not duplicated. These entities are separately updated in *Phase 3*.

In order to ensure symmetrical behavior among partitions, we add two constraints to the original serial algorithm. The first constraint requires that, for any partition in which a

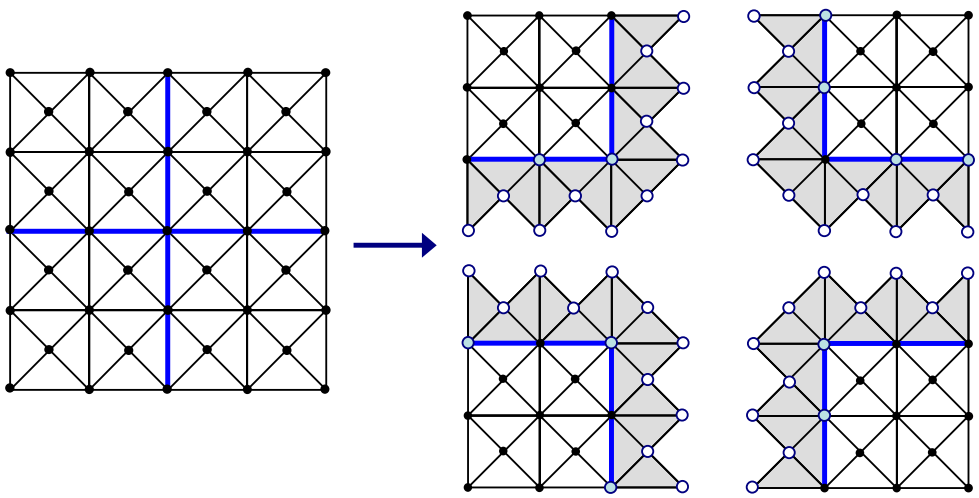


Fig. 8 Sample mesh used to illustrate the algorithm for parallel insertion of cohesive elements. The original mesh has been decomposed into four distinct partitions

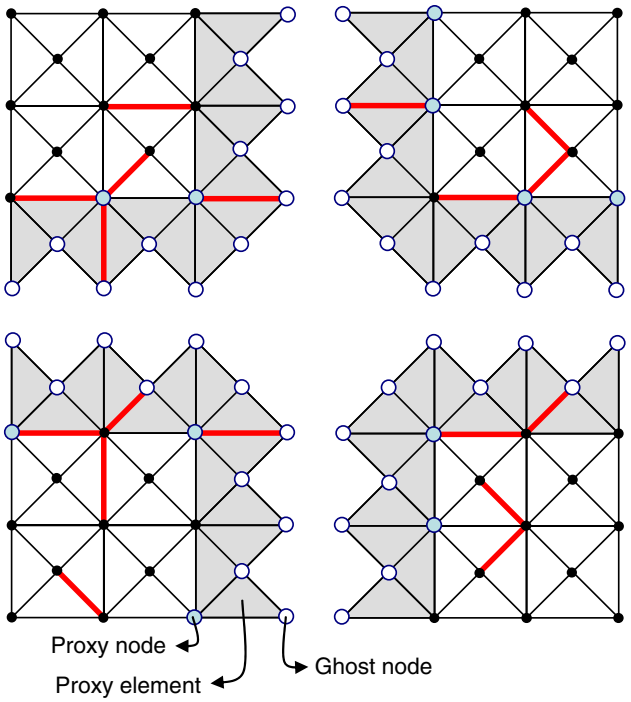


Fig. 9 Fractured facets are highlighted on the distributed mesh. The determination of those facets is done by the analysis application. Proxy elements and nodes are represented by *shaded shapes*, while ghost nodes are represented by *hollow circles*

newly created node is present, the node has a reference to the same element (in TopS, and thus ParTopS, every node holds a reference to one of the elements that is connected to it). This can be achieved by using a uniform criterion for selecting that element. For simplicity, we choose, among the elements currently adjacent to the node, the one with the smallest tuple: (*owner_partition, owner_handle*). Tuples are compared in lexicographic order, first by *owner_partition* and next by *owner_handle*. That means that the values of *owner_partition* are first compared; if they are equal, the values of *owner_handle* are compared.

The second constraint requires that all the copies of a new node or a cohesive element are assigned to the same owner partition. In the case of a node, the second constraint is achieved by assigning it to the partition of the referenced element. As all the copies of the node refer to the same real element (according to the first constraint), then they are naturally assigned to the same owner partition. We assign cohesive elements to the partition of one of its adjacent elements. In this case, the representative element is selected based on the criterion applied for the first constraint (smallest tuple (*owner_partition, owner_handle*)). The anchors that define implicit entities can also be updated using this same procedure.

Phase 1 of the algorithm is illustrated in Fig. 10 for two neighboring partitions. Three cohesive elements are simultaneously inserted around a node that is present in both partitions (see Fig. 10a). Although the order of insertion is different in each partition, the same topological results are achieved in the end (see Fig. 10c). Note that the same elements are referenced by each of the newly created nodes in either partition (as indicated by the "*x*" marks in Fig. 10), as a result of the two topological criteria described above.

6.5 Updating new proxy entities (Phase 2)

After *Phase 1*, the topology of local and proxy entities is ensured to be consistent. However, references from new proxy entities (cohesive elements and duplicated nodes) to the corresponding real entities still have to be computed (see Fig. 11). Although the owner partition of a new entity can be

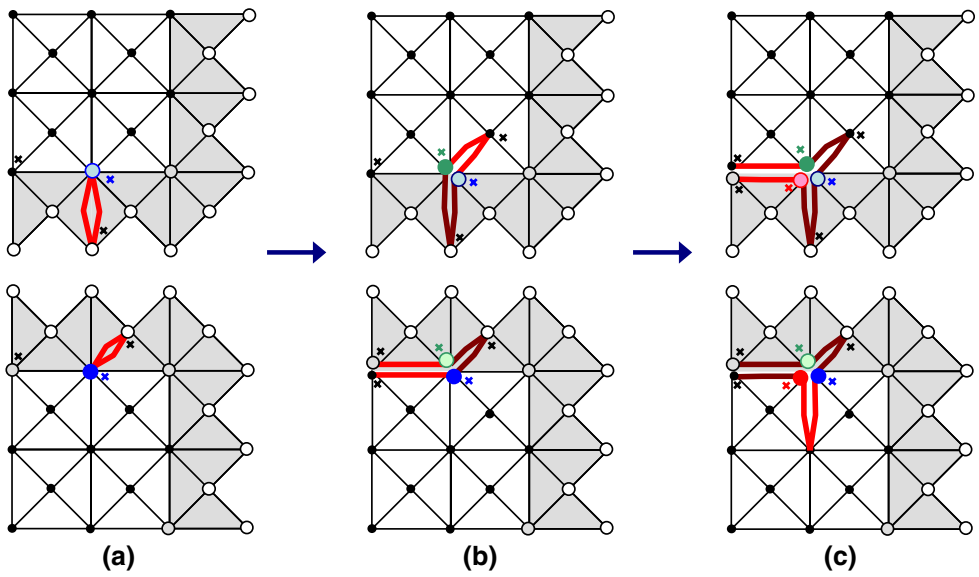


Fig. 10 Execution of *Phase 1* of the algorithm. Three cohesive elements are inserted, in different orders, around a node present in two neighboring partitions (**a**–**c**). In spite of the order in which the elements are inserted, symmetrical topological results are obtained at local and proxy entities in both partitions (**c**). The "*x*" marks indicate the element referenced by each modified node. Ghost nodes are not duplicated. They will be updated in *Phase 3* of the algorithm

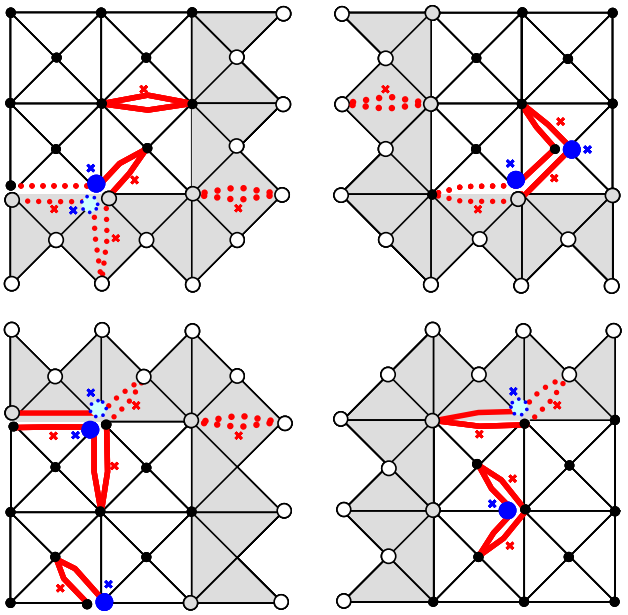


Fig. 11 The complete mesh after *Phase 1*. The partition chosen as the owner of a new cohesive element or node is the owner of one of its incident elements (as indicated by "*x*" marks). The new cohesive elements and nodes are highlighted. Respective proxy entities are represented with dotted lines. Although the topology of proxy and local entities is ensured to be consistent at this time, the references to the corresponding real entities are still missing

inferred directly during *Phase 1*, as it is the partition of one of the elements adjacent to the entity, the handle of the entity with respect to the owner partition is not known. This information must be requested from the owner partition.

Phase 2 is responsible for updating the missing references of new proxy cohesive elements and nodes, and works as follows: a list of requests for the references is sent to each neighboring partition, which accesses the respective real entities and replies with the corresponding handles. Then, current partition updates the references of the new entities with the received values.

Note that we cannot use the new proxy entities to request their own missing references. We observe, though, that any element (cohesive or not) can be uniquely identified by one of its adjacent elements. This is expressed by the tuple: (*owner_partition_adj*, *owner_handle_adj*, *lid*), where *owner_partition_adj* is the *id* of the partition that owns the adjacent element, *owner_handle_adj* is the local handle of the adjacent element in that partition, and *lid* is the local *id* of the facet-use of the adjacent element that is incident to the current one. Correspondingly, a node can also be uniquely identified by one of its incident elements and its local *id* with respect to that element, as illustrated by the markers in Fig. 11.

Therefore, adjacent elements are used to obtain missing references of new proxy entities. The owner partition of a cohesive element is the same partition as one of its adjacent

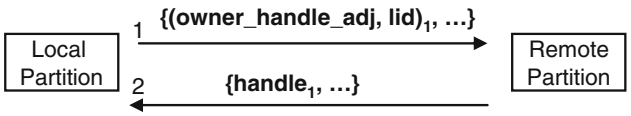


Fig. 12 Messages exchanged between local and remote partitions in order to update the references from new proxy entities to the corresponding real entities

elements (see Sect. 6.4). Thus, the representative adjacent element is used for requesting the reference of the cohesive element. Similarly, the element referenced by a new proxy node is used for requesting the missing reference from this node to the corresponding real entity (as, by definition, the node is owned by the same partition as that element). Note that *Phase 1* does not change the set of neighboring partitions.

The messages exchanged between two partitions to update proxy references are illustrated in Fig. 12. Local partition sends a request message, consisting of a list of tuples: (*owner_handle_adj*, *lid*), to each remote partition (*owner_partition_adj*) that owns the corresponding adjacent elements. The reply message consists of the handles of the required entities with respect to the owner partition.

6.6 Updating affected ghost entities (Phase 3)

The last phase of the algorithm is responsible for updating ghost nodes that have been affected by duplication of nodes at other partitions, due to the insertion of new cohesive elements (see Fig. 13). As ghost nodes lay on the

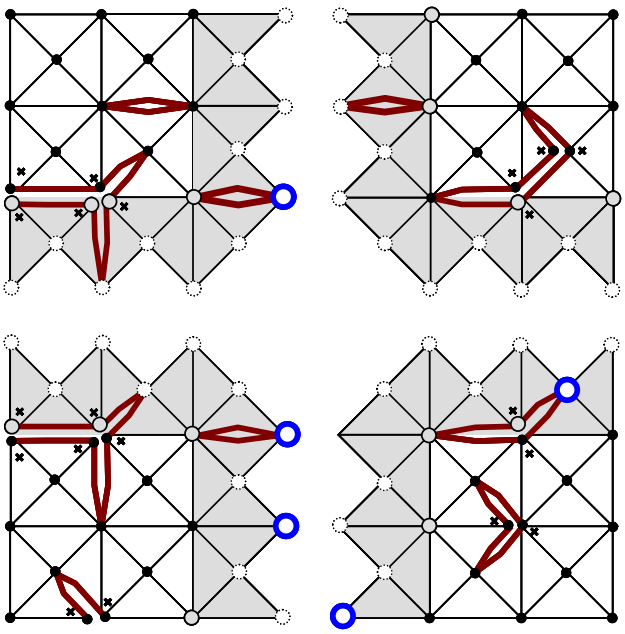


Fig. 13 The complete mesh before *Phase 3*. The highlighted ghost nodes were duplicated in other partitions when new cohesive elements were inserted, and thus need to be updated

boundaries of the communication layer of each partition, the ones that have changed cannot be determined based only on local information. For example, a cohesive element may have been inserted at a remote facet that is not known by the current partition, resulting in the duplication of the real node corresponding to a ghost node in the current partition. In Fig. 13, ghost nodes affected by duplication of nodes from other partitions are shown. In order to keep mesh topology consistent, the update procedure for a ghost node may either have to replace the reference to the corresponding real entity or to split the node into two or more new ones (see Fig. 13).

The simplest way to keep topology of ghost nodes consistent is to update all of them. However, it is common that few cohesive elements are inserted during each simulation step. Thus, we prefer an approach that is proportional to the number of affected entities. Then, each partition that owns duplicated nodes that may affect ghost nodes in other partitions is responsible for sending notification messages.

However, in ParTopS, real entities do not keep a list of references to their remote proxies or ghosts. In that manner, we avoid the overhead of storing and maintaining dynamic reference lists. On the other hand, this constrains all the communication related to a real entity to be initiated by the partitions that have a proxy or ghost of the entity (as real entities do not know of the existence of associated proxies or ghosts). Thus, in order to allow a partition to efficiently notify other partitions of node duplications, we need a procedure to identify which real entities might have associated ghosts, without explicitly maintaining dynamic reference lists.

In this work, instead of storing for each local node the list of corresponding ghosts, we have opted for a per-element approach: each real element has references to the partitions of its proxies. Thus, elements with proxies in other partitions are responsible for notifying changes on ghost nodes and transfer required data. The advantage of this approach is that existing references to proxies do not have to be updated when new cohesive elements are inserted, because no proxy elements are replaced or removed from the mesh during this process. In addition, communication can be entirely done through proxy elements. Therefore, a partition holding a ghost node does not have to be considered as neighbor of the partition with the corresponding real entity. Then, two partitions are only considered neighbors if one has a proxy for an entity from the other. This is consistent with the definition of partition neighborhood introduced in Sect. 5.2. As existing proxies are not replaced or removed, and new cohesive elements are assigned to the same partition as one of their adjacent elements, neighborhood sets of partitions do not have to be updated.

Topological changes on a node affect its incident elements because they hold a reference to the node. Consequently, when a ghost node is modified, incident proxy elements are also affected (notice that ghost nodes can only be incident to proxy elements). If a local node is created when a cohesive element is inserted and the node is incident to elements with proxies in other partitions, then it is possible that the node has corresponding ghosts in those partitions. Hence, the ghost nodes and the affected proxy elements should be updated accordingly. Thus, affected proxy elements are notified by the corresponding real elements in order to update all of their ghost nodes. In this manner, we ensure that modified ghost nodes and incident proxy elements are kept up-to-date.

As discussed in Sect. 6.5, a node can be uniquely identified by one of its adjacent elements and its local *id* in the element. Consequently, we can use proxy elements to request data for the incident ghost nodes. In this case, however, requests are sent to the owner partition of the proxy elements, rather than the owners of the ghost nodes. Thus, the corresponding nodes may be represented as proxies in the owner partition of the proxy elements, and the retrieved data might be outdated regarding the real nodes when requested by the local partition. Nevertheless, modified proxy entities were updated in *Phase 2*, and then ghost data received through incident proxy elements will be consistent during *Phase 3*.

This approach has one drawback, which is discussed next. With per-element notifications, all the ghost nodes of a proxy element that received a notification are updated (as the element does not know which ones have actually changed). Although this results in a few unnecessary updates, the number tends to be small and the procedure still remains proportional to the amount of duplicated nodes.

For the purpose of the present distributed data structure, rather than storing lists of all the partitions that have proxies for a given local element, we use a single integer to indicate the existence of those proxies. If there is only one partition with a proxy for the element, which is the most common case, the integer value is the *id* of the partition. If there are two or more partitions, a negative value is used to indicate the number of such partitions. In this case, notifications must be sent to all the neighboring partitions. If no partition has a proxy for the element, the value of the integer is zero.

The procedure for updating ghost nodes starts by determining the local elements with proxies in other partitions that are incident to newly duplicated nodes (see Fig. 14). Although several of the elements incident to a node may have a proxy in the same partition, only one per-partition has to be selected. Then, a notification message is sent to the partitions containing the proxy elements. This

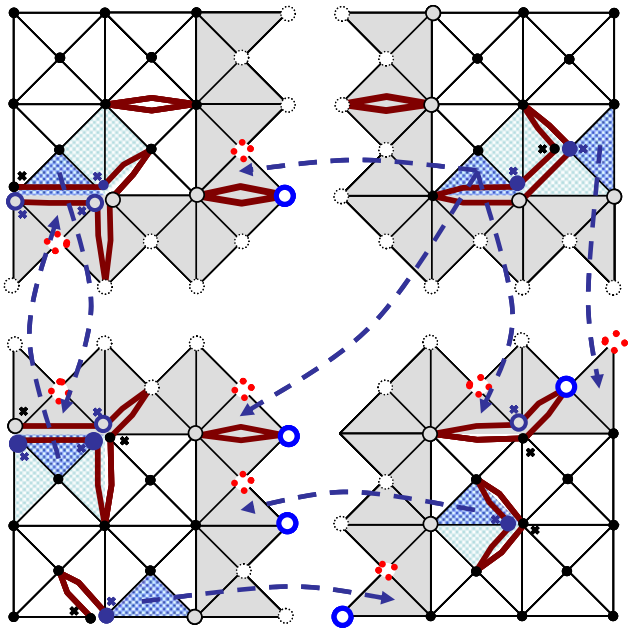


Fig. 14 Local elements that are incident to duplicated nodes and have a corresponding proxy element in another partition are emphasized. These elements notify of nodal duplications that may affect ghost nodes. However, only one element per duplicated node and remote partition need to be used (highlighted elements and *dashed arrows*). All the ghost nodes of notified proxy elements are marked as *outdated*. The highlighted ghost nodes with *solid lines* have actually changed and must be updated, whereas the ones with dotted lines were marked as a consequence of the per-element notification scheme used

message consists of the handles of the selected local elements (see Fig. 17).

When the notification message is received by a partition, the handles are mapped to the corresponding proxy elements and all their ghost nodes are marked as *outdated*. For each proxy element that is incident to an outdated ghost node (see Fig. 15), a request for the corresponding nodal data is sent to the owner partition of the element. Requests consist of the tuple: (*owner_partition*, *owner_handle*, *lid*), where *owner_partition* is the *id* of the owner partition of the element, *owner_handle* is the handle of the element with respect to the owner partition, and *lid* is the local *id* of the node in the element. Note that more than one request for the same node may be sent, as several elements may be incident to that node. However, with this approach, the duplication of ghost nodes is handled naturally and in a consistent way within each incident proxy element. The remote partitions then reply with the up-to-date nodal data (see Fig. 17), and the ghost nodes are replaced. After ghost nodes have been updated, the anchors of the incident implicit entities are adjusted so that local topological consistency is maintained. The final mesh configuration is shown in Fig. 16.

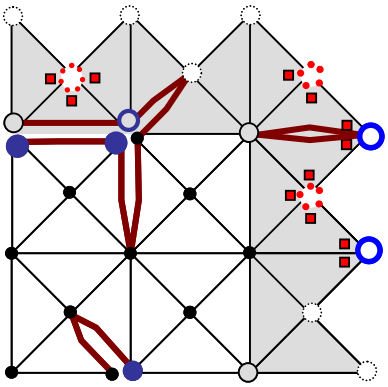


Fig. 15 For each proxy element incident to a ghost node marked as outdated, a request for the corresponding node is sent to the partition that owns the element. Node requests are indicated by *square marks* besides ghost nodes

The messages exchanged between two partitions to update ghost nodes are summarized in Fig. 17. The local partition receives notification messages consisting of handles (*owner_handle*) of remote real elements corresponding to local proxies. The handles are mapped to respective

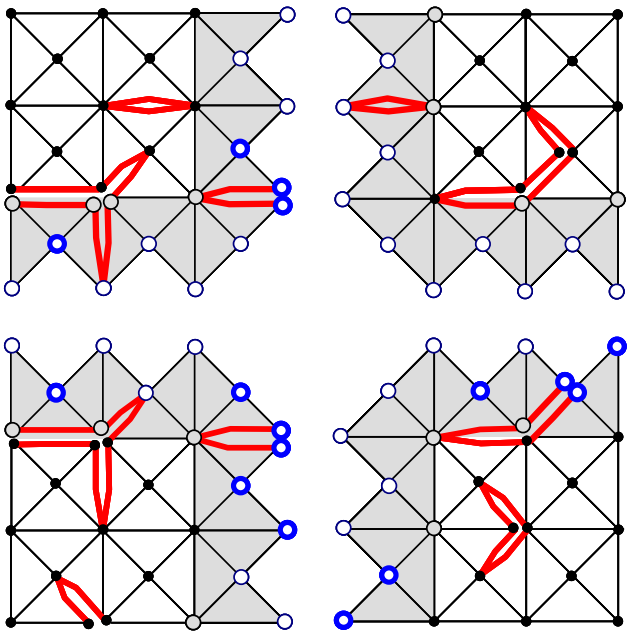


Fig. 16 The final mesh configuration with cohesive elements and up-to-date ghost nodes

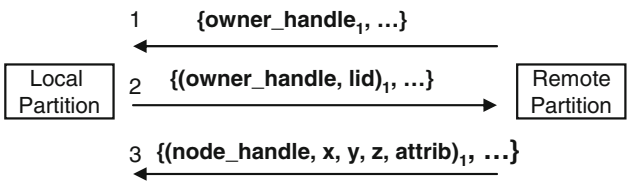


Fig. 17 Messages exchanged between local and remote partitions in order to update ghost entities affected by duplication of remote nodes

proxy elements and all their ghost nodes are marked as *outdated*. A request message, consisting of tuples: (*owner_handle*, *lid*), is sent to each remote partition that owns proxy elements incident to outdated ghost nodes. Remote partitions then reply with up-to-date nodal data: (*node_handle*, *x*, *y*, *z*, *attributes*).

7 Computational experiments

In order to test the present topological framework for parallel fragmentation, we have run a set of computational experiments with meshes of different types of elements, both in 2D and in 3D, linear and quadratic. In the experiments, the insertion of cohesive elements is decoupled from the mechanics analysis. The goal is to verify the proposed data structure and algorithm as a valid topological framework for parallel fragmentation simulation.

The two basic models used in the experiments are illustrated in Fig. 18. The two-dimensional model consists of a simple Cartesian grid of ($nx \times ny$) quadrilateral cells decomposed into four triangles each. Equivalently, the three-dimensional model consists of a grid of ($nx \times ny \times nz$) hexahedral cells decomposed into six tetrahedra each. In order to run in parallel, those models are divided into a set of distinct partitions, each representing an individual processing unit.

For each mesh type, the correctness and efficiency of the proposed topological framework was tested by varying the following parameters: mesh discretization, number of partitions, and number of processors. In each experiment, cohesive elements were randomly inserted at about 50% of the total internal facets (not on the boundary) of each partition of the mesh. This results in arbitrary and complex crack patterns. Additionally, in order to mimic the behavior of real fragmentation simulations, in which the number of new cohesive elements at each simulation step is usually much smaller than the total number of facets, we have inserted the cohesive elements in an incremental way: at each iteration, cohesive elements are inserted at 1% of the internal facets. At the end of each step, partitions are

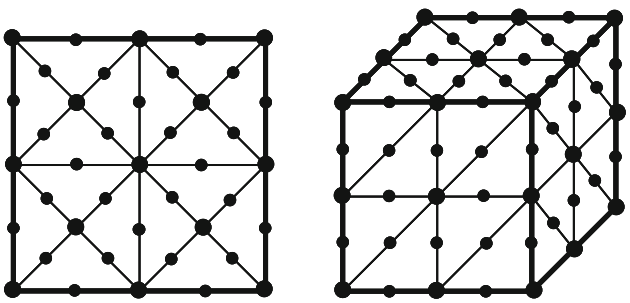


Fig. 18 Examples of basic mesh templates used to validate the proposed parallel framework

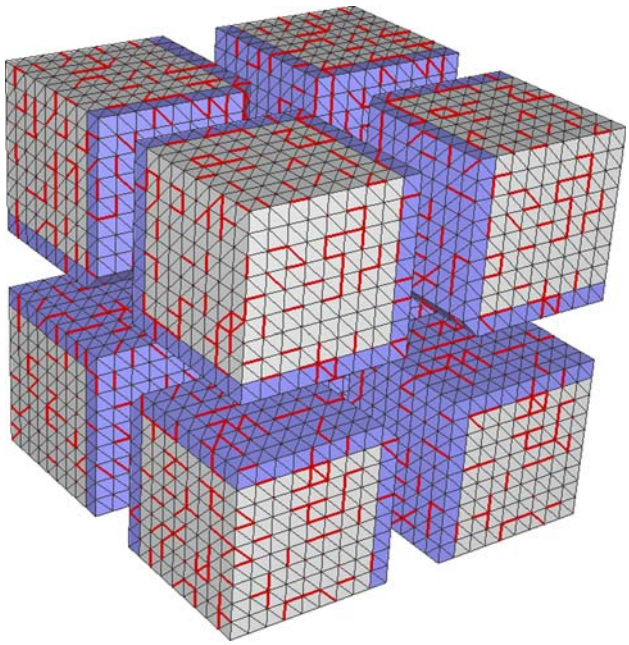


Fig. 19 Distributed linear tetrahedral (TET4) mesh with $16 \times 16 \times 16$ discretization after randomly inserting cohesive elements at about 10% of the internal facets. Element insertion was incrementally performed at a rate of 1% of the facets per iteration. Communication layer and cohesive elements are emphasized

synchronized as described: new proxies (cohesive elements and nodes) have their references updated and modified ghost nodes are re-assigned. After 50 steps, the total number of inserted cohesive elements is about 50% of the total number of internal facets of the initial model. In Fig. 19, a linear tetrahedral mesh that was used in the experiments is presented, after inserting cohesive elements at about 10% of the internal facets.

The experiments were executed on a cluster of 12 machines connected by a Gigabit Ethernet network. Each machine has an Intel(R) Pentium(R) D processor 3.40 GHz (two cores) and 2 GB of RAM. The operating system is Red Hat Linux 3.4.6-9, with a 32-bit kernel, and the compiler is gcc v. 3.4.6.

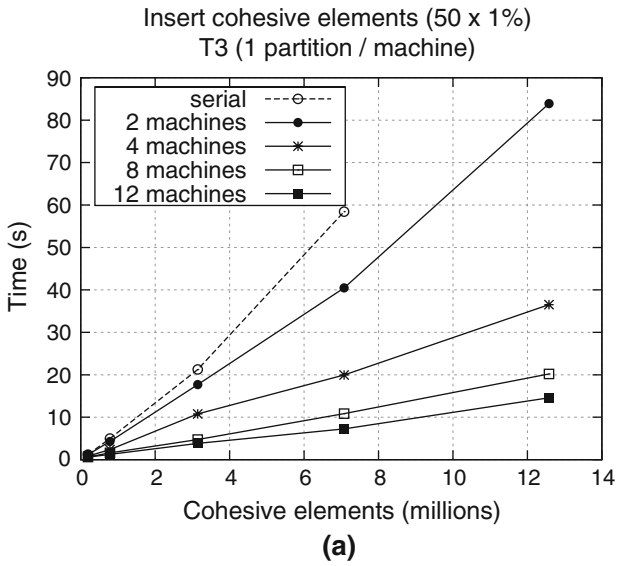
The results achieved are summarized in Tables 1 and 2. Table 1 shows the various meshes used in the experiments and the corresponding average serial times for inserting cohesive elements at 50% of the internal facets of each model. The largest models did not fit in the memory of one machine. Table 2 presents the results of the parallel algorithm executed with varying number of machines, and for one and two mesh partitions per machine (each partition associated to a processor core), respectively. Parallel times correspond to average times of running each experiment 5 times. The reported times correspond to the total time needed to execute all the 50 steps of cohesive element insertion. At the end, the total number of cohesive elements is about the same as in the serial experiment.

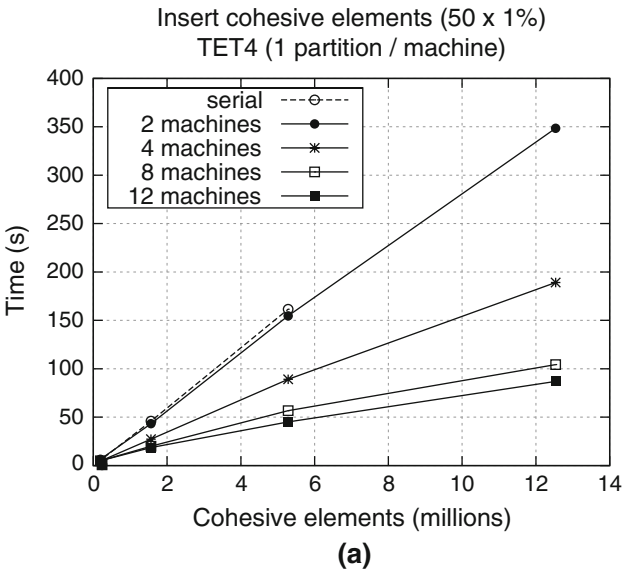
Table 1 Times, in seconds, for serial insertion of cohesive elements in various meshes

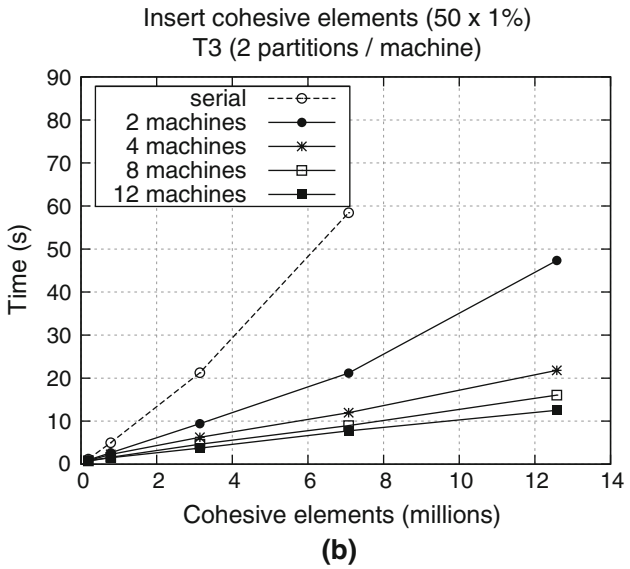
Element type	Mesh discretization	# bulk elements	# nodes	# cohesive elems inserted	Time (s)
T3	256 × 256	262,144	131,585	196,352	1.20
	512 × 512	1,048,576	525,313	785,920	4.99
	1,024 × 1,024	4,194,304	2,099,201	3,144,704	21.24
	1,536 × 1,536	9,437,184	4,721,665	7,076,352	58.43
	2,048 × 2,048	16,777,216	8,392,705	12,580,864	n/a
T6	256 × 256	262,144	525,313	196,352	1.42
	512 × 512	1,048,576	2,099,201	785,920	5.99
	1,024 × 1,024	4,194,304	8,392,705	3,144,704	30.19
	1,536 × 1,536	9,437,184	18,880,513	7,076,352	n/a
	2,048 × 2,048	16,777,216	33,562,625	12,580,864	n/a
TET4	16 × 16 × 16	24,576	4,913	23,808	0.80
	32 × 32 × 32	196,608	35,937	193,536	5.39
	64 × 64 × 64	1,572,864	274,625	1,560,576	45.87
	96 × 96 × 96	5,308,416	912,673	5,280,768	161.52
	128 × 128 × 128	12,582,912	2,146,689	12,533,760	n/a
TET10	16 × 16 × 16	24,576	35,937	23,808	0.93
	32 × 32 × 32	196,608	274,625	193,536	6.21
	64 × 64 × 64	1,572,864	2,146,689	1,560,576	53.55
	96 × 96 × 96	5,308,416	7,189,057	5,280,768	n/a
	128 × 128 × 128	12,582,912	16,974,593	12,533,760	n/a

Table 2 Times, in seconds, for the parallel insertion of cohesive elements in various meshes

Element type	Mesh discretization	Time (s) serial	Time (s)—parallel							
			Number of machines (one partition per machine)				Number of machines (two partitions per machine)			
			2	4	8	12	2	4	8	12
T3	256 × 256	1.20	1.17	0.84	0.62	0.58	0.97	0.84	0.78	0.77
	512 × 512	4.99	4.30	2.37	1.60	1.26	2.66	2.28	1.62	1.49
	1,024 × 1,024	21.24	17.68	10.79	4.75	3.83	9.41	6.25	4.65	3.72
	1,536 × 1,536	58.43	40.48	19.96	10.86	7.27	21.16	11.98	8.96	7.75
	2,048 × 2,048	n/a	83.91	36.52	20.20	14.57	47.35	21.80	16.05	12.51
T6	256 × 256	1.42	1.42	0.86	0.68	0.64	1.03	0.95	0.84	0.84
	512 × 512	5.99	5.32	2.88	1.75	1.50	3.27	2.68	1.90	1.69
	1,024 × 1,024	30.19	21.88	11.04	5.96	4.38	11.96	7.23	5.49	4.60
	1,536 × 1,536	n/a	n/a	25.07	13.45	9.72	47.46	16.29	9.90	8.58
	2,048 × 2,048	n/a	n/a	56.29	24.40	18.77	n/a	42.17	18.31	15.07
TET4	16 × 16 × 16	0.80	0.94	0.89	0.71	0.83	0.85	0.92	1.19	1.62
	32 × 32 × 32	5.39	6.57	4.66	4.92	4.99	4.71	4.48	4.85	4.96
	64 × 64 × 64	45.87	43.30	27.31	19.95	18.64	30.83	22.26	19.61	19.61
	96 × 96 × 96	161.52	154.74	89.13	56.82	45.17	91.44	61.09	45.58	39.46
	128 × 128 × 128	n/a	348.55	189.11	104.29	87.05	199.48	113.92	79.99	69.45
TET10	16 × 16 × 16	0.93	1.34	1.36	1.30	1.14	1.17	1.35	1.72	1.97
	32 × 32 × 32	6.21	7.54	6.43	7.10	7.52	6.35	6.72	8.34	8.64
	64 × 64 × 64	53.55	56.07	36.61	26.58	25.39	39.98	29.91	27.47	28.09
	96 × 96 × 96	n/a	192.18	111.65	73.61	60.98	122.28	79.53	63.26	54.00
	128 × 128 × 128	n/a	n/a	266.16	137.73	117.47	n/a	183.68	110.59	92.66







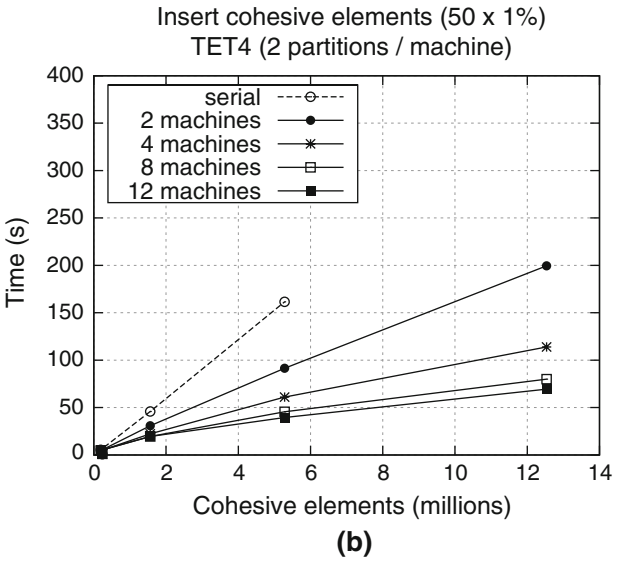


Fig. 20 Time versus average number of cohesive elements inserted for meshes of T3 elements with varying number of machines: **a** one mesh partition per machine; **b** two mesh partitions per machine. Note that the results for the 2,048 × 2,048 mesh discretization could not be computed for the serial algorithm

Fig. 21 Time versus average number of cohesive elements inserted for meshes of TET4 elements with varying number of machines: **a** one mesh partition per machine; **b** two mesh partitions per machine. Note that the results for the 128 × 128 × 128 mesh discretization could not be computed for the serial algorithm

In Figs. 20 and 21, we plot time versus approximate total number of cohesive elements inserted in linear triangular (T3) and tetrahedral (TET4) meshes, with the parallel and serial versions of the algorithm. As can be noted, in these experiments, when using four or more machines, the time spent by the parallel algorithm scaled approximately linearly with the number of cohesive elements inserted. Linear behavior was expected because the time required for the serial insertion of cohesive elements in each partition (*Phase 1*) is proportional to the number of new elements (see Ref. [13]) and communication is proportional to the number of affected entities near partition boundaries (number of new proxy cohesive elements and modified ghost nodes). The serial algorithm and the parallel version with only two machines did not scale linearly probably because of substantial use of the available memory.

In Fig. 22, we plot the time spent to insert all cohesive elements versus the number of machines used for the T3 (with 1,536 × 1,536 discretization) and TET4 (with 96 × 96 × 96 discretization) meshes. Those two mesh discretizations were chosen because they are the largest ones for which the serial algorithm could be executed in our computational environment. The results are compared to

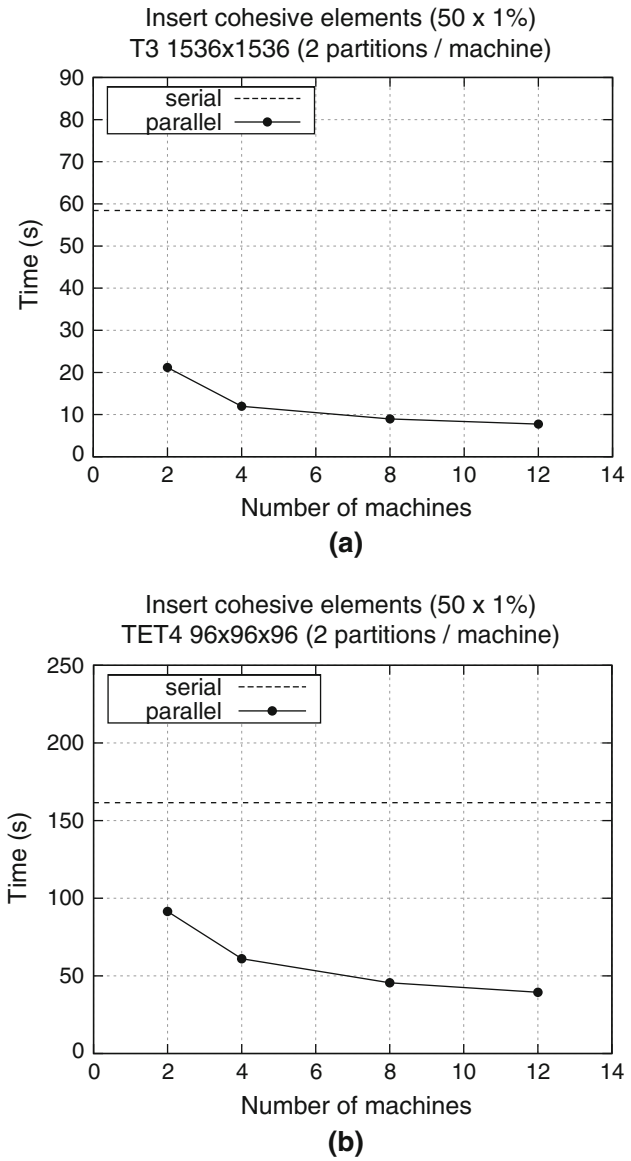


Fig. 22 Time for the parallel insertion of cohesive elements versus number of machines considering two partitions per machine: **a** mesh of T3 elements with 1,536 × 1,536 discretization; **b** mesh of TET4 elements with 96 × 96 × 96 discretization

the time required by the serial algorithm. For these mesh sizes, the most significant performance gains occurs when up to four machines are used. The addition of new machines does not yield proportional performance benefits, as communication costs tend to be higher while less processing is done per machine.

The proportional time spent at each step of the parallel algorithm is depicted in Fig. 23. The figure illustrates the time spent by inserting 1% of the facets in the TET4 meshes, with four machines and two partitions per machine. It can be seen that proportional time for the network communication (*Phase 2* + *Phase 3*) decreases inversely with the size of the mesh. This is natural because

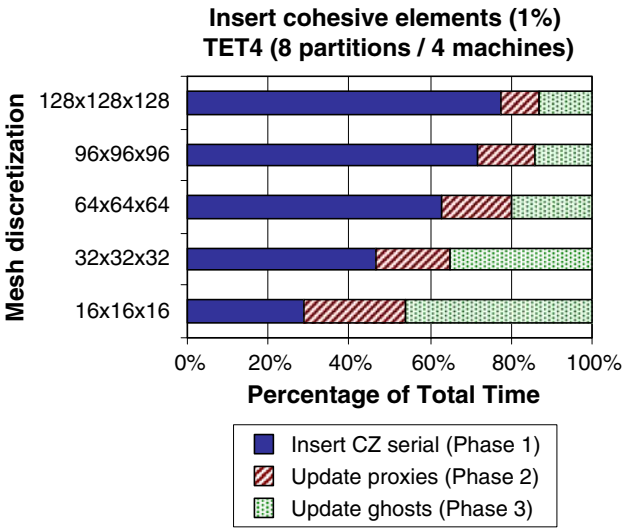


Fig. 23 Percentage of the total time required by each step of the proposed algorithm when cohesive elements are inserted at 1% of the facets in a TET4 mesh. Various mesh discretizations are presented

the communication cost is proportional to the number of cohesive elements inserted along the partition's interfaces, while the serial cost (within each partition) is proportional to the total number of inserted cohesive elements. In that manner, the performance penalty of the parallel implementation, due to the need of synchronization, is substantially reduced if larger meshes are used.

To address the practical application of ParTopS, Fig. 24 presents two unstructured tetrahedral meshes decomposed into three (Fig. 24a) and four domains (Fig. 24b). Cohesive elements were randomly inserted at about 10% of the total number of internal facets. Communication layers and cohesive elements are emphasized in the figure.

The main purposes of creating a framework for supporting parallel finite element analysis are: (1) to handle large models; (2) to improve the performance of model computation. The achieved results demonstrate that our proposal meets both goals. Besides enabling the insertion of cohesive elements in meshes that could not be fit within a single machine memory, the proposed parallel algorithm resulted in considerable performance gains for inserting cohesive elements in those meshes, if compared with the serial version of the algorithm.

7.1 Scalability issues

Scalability of a parallel system is related to how efficiently an increasing number of resources may be used to solve larger complex problems [19]. Several approaches have been developed to measure scalability under different circumstances [19, 38, 39]. In this section, we use the isoefficiency metric presented in [40] to estimate the scalability of the proposed algorithm for 2D test models. This metric

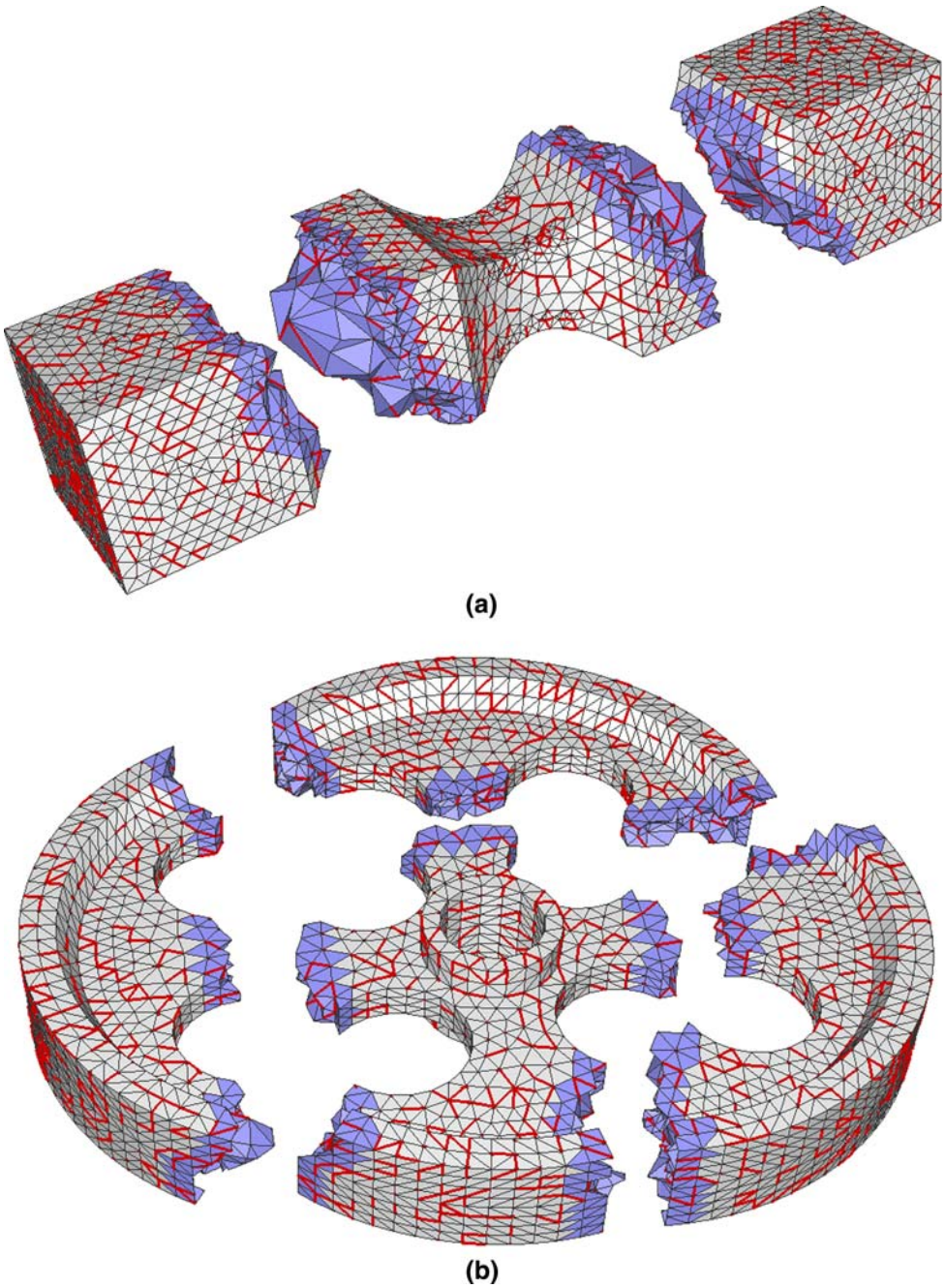


Fig. 24 Two unstructured tetrahedral meshes (**a**, **b**) used to test the proposed algorithm. Cohesive elements were randomly inserted at about 10% of the total number of internal facets. Element insertion was incrementally performed at a rate of 1% of the internal facets per iteration. Communication layers and cohesive elements are emphasized

relates the problem size to the number of processors necessary to maintain the efficiency of a system. Similar results can be obtained for 3D models.

Consider the execution time $T(n, p)$ of a parallel system with problem size n on p processors. The sequential time is $T_1 = T(n, 1)$, and $T_0 = T_0(n, p)$ is the total overhead time introduced due to the parallel implementation. Then, the efficiency of the system [40] can be defined as: $E = 1/(1 + T_0/T_1)$, or equivalently: $T_1 = (E/(1 - E))T_0 = KT_0$.

In order to maintain efficiency, the time for the serial computation T_1 must increase in a rate that is greater than or equal to the parallel overhead T_0. Hence, $T_1 \geq KT_0$. As shown in Ref. [13], the serial time for insertion of cohesive elements scales approximately linearly with the number of elements inserted. Thus, for the test meshes in which n cohesive elements have been inserted, the sequential execution time can be expressed as $T_1 = nt_c$, where t_c is the average cost per operation. In that manner, efficiency is maintained if $n \geq CT_0$, where $C = K/t_c$. Parallel overhead T_0 is due to replicated computation at communication layer and messages exchanged between neighboring partitions, as illustrated in Fig. 25. We observe that the amount of

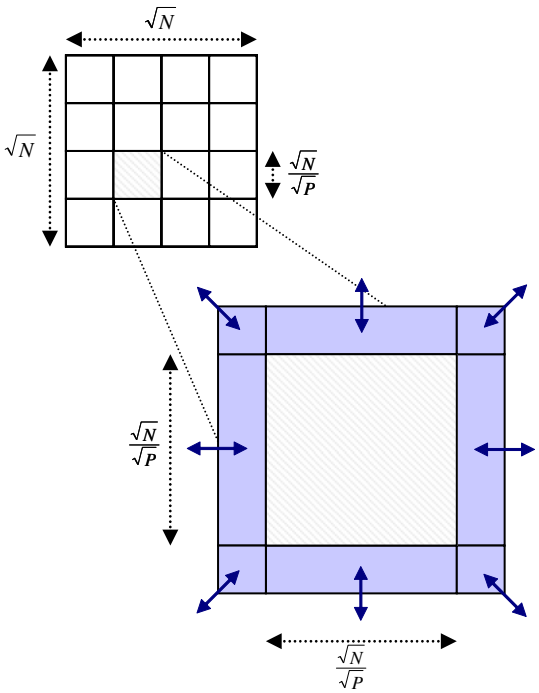


Fig. 25 Sample 2D mesh with n elements divided into p partitions, and associated communication pattern of a single mesh partition

replicated computation is proportional to the size of the communication layer, and thus to $\sqrt{n}/\sqrt{p}$. The number of messages exchanged between neighboring partitions during the execution of the algorithm is constant and the size of each message is proportional to the size of the communication layer. The parallel overhead per processor can be expressed as: $T_p \approx c_1(\sqrt{n}/\sqrt{p})t_c + c_2 t_s + c_3(\sqrt{n}/\sqrt{p})t_w$, where c_1, c_2 and c_3 are constant multipliers, t_s is the average startup time of a message and t_w is the average time to send a unit of data. Then, the total overhead is $T_0 = pT_p = c_1 t_c\sqrt{np} + c_2 t_s + c_3 t_w\sqrt{np}$. We assume that load is balanced among partitions and thus processor idle times are not expected to be significant.

For a large number of processors, efficiency is maintained when the problem size n asymptotically grows at least as fast as T_0. With the growing rate of T_0 proportional to $\sqrt{np}$, we have the following relation: $n \geq C\sqrt{np}$, which yields $n \geq C^2 p$. Therefore, when n grows proportionally to p, efficiency of the algorithm is maintained, and thus the algorithm is expected to scale linearly with the number of processors.

8 Concluding remarks

ParTopS is a parallel topological framework for moving boundary simulations and fragmentation. In order to provide support for parallel topological operations, we have extended the serial topological data structure named TopS [4, 12, 13] for distributed mesh representation. A minimum amount of data was added to the data structure so that the compactness and reduced representation of TopS was maintained. The proposed framework includes a parallel algorithm for dynamic insertion of cohesive elements that is based on the criteria for fracture facet classification introduced by Paulino et al. [13], and thus, can be applied in a uniform way to different types of finite element meshes. The time spent to update the distributed data structure is expected to be proportional to the number of cohesive elements inserted in the mesh.

A set of computational experiments that demonstrate the correctness of the proposed topological framework was executed. The achieved results have shown that the framework is capable of handling relatively large models, while presenting significant performance gains for inserting cohesive elements in those models. The present topological framework represents an important step towards the ability to perform moving boundary and extrinsic fragmentation simulation of massive models.

In the present work, ParTopS has been implemented on top of Charm++. However, the ParTopS algorithm is general and could be implemented directly in another parallel environment (e.g. MPI) without any difficulty. The current implementation covers the representation of entities and the corresponding procedures needed for parallel insertion of cohesive elements. To extend its range of application to other types of adaptive analyses (e.g. h-version), it may be necessary to represent additional information. We also expect to extend ParTopS with additional topological operators to support general mesh modifications, including "on-demand" refinement and coarsening.

Acknowledgments RE and WC would like to thank the Tecgraf laboratory at PUC-Rio, which is mainly funded by the Brazilian oil company, Petrobras. RE and NR also thank CNPq (National Council for Scientific and Technological Development of Brazil) for partially funding this research. GHP acknowledges support from the National Science Foundation (NSF) through Grant CMMI #0800805. The information presented in this paper is the sole opinion of the authors and does not necessarily reflect the views of the sponsoring agencies.

References

1. Zhang Z, Paulino GH, Celes W (2007) Extrinsic cohesive modelling of dynamic fracture and microbranching instability in brittle materials. Int J Numer Methods Eng 72(8):893–923
2. Papoulia KD, Vavasis SA, Ganguly P (2006) Spatial convergence of crack nucleation using a cohesive finite-element model on a pinwheel-based mesh. Int J Numer Methods Eng 67(1):1–16
3. Beall MW, Shephard MS (1997) A general topology-based mesh data structure. Int J Numer Methods Eng 40(9):1573–1596
4. Celes W, Paulino GH, Espinha R (2005) A compact adjacency-based topological data structure for finite element mesh representation. Int J Numer Methods Eng 64(11):1529–1565

5. Garimella RV (2002) Mesh data structure selection for mesh generation and FEA applications. Int J Numer Methods Eng 55(4):451–478
6. Park K, Paulino GH, Roesler JR (2009) A unified potential-based cohesive model of mixed-mode fracture. J Mech Phys Solids. doi: 10.1016/j.jmps.2008.10.003 (in press)
7. Owen SJ, Shephard MS (2003) Editorial: special issue on trends in unstructured mesh generation. Int J Numer Methods Eng 58(2):159–160
8. Pandolfi A, Ortiz M (1998) Solid modeling aspects of three-dimensional fragmentation. Eng Comput 14(4):287–308
9. Pandolfi A, Ortiz M (2002) An efficient adaptive procedure for three-dimensional fragmentation simulations. Eng Comput 18(2):148–159
10. Paulino GH, Jin Z-H, Dodds RH Jr (2003) Failure of functionally graded materials. In: Karihaloo B, Knauss WG (eds) Comprehensive structure integrity, 2(13) edn. Elsevier, Amsterdam, pp 607–644
11. Zhang Z, Paulino GH (2005) Cohesive zone modeling of dynamic failure in homogeneous and functionally graded materials. Special Issue on Inelastic Response of Multiphase Materials. Int J Plast 21(6):1195–1254
12. Celes W, Paulino GH, Espinha R (2005) Efficient handling of implicit entities in reduced mesh representations. J Comput Inf Sci Eng 5(4):348–359 (Special Issue on Mesh-Based Geometric Data Process)
13. Paulino GH, Celes W, Espinha R, Zhang Z (2008) A general topology-based framework for adaptive insertion of cohesive elements in finite element meshes. Eng Comput 24(1):59–78
14. Kalé LV, Krishnan S (1993) CHARM++: a portable concurrent object oriented system based on C++. In: Paepcke A (ed) Proceedings of OOPSLA'93. ACM Press, September 1993, pp 91–108
15. Kalé LV, Krishnan S (1996) Charm++: Parallel Programming with Message-Driven Objects. In: Wilson GV, Lu P (eds) Parallel Programming using C++. MIT Press, London, pp 175–213
16. Lawlor O, Chakravorty S, Wilmarth T, Choudhury N, Dooley I, Zheng G, Kalé L (2006) ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications. Eng Comput 22(3–4):215–235
17. Remacle J-F, Klaas O, Flaherty JE, Shephard MS (2002) Parallel algorithm oriented mesh database. Eng Comput 18(3):274–284
18. Seol ES, Shephard MS (2006) Efficient distributed mesh data structure for parallel automated adaptive analysis. Eng Comput 22(3–4):197–213
19. Foster I (1995) Designing and building parallel programs: concepts and tools for parallel software engineering. Addison–Wesley, Boston
20. Waltz J (2002) Derived data structure algorithms for unstructured finite element meshes. Int.J Numer Methods Eng 54(7):945–963
21. Gara A, Blumrich MA, Chen D et al (2005) Overview of the Blue Gene/L system architecture. IBM J Res Dev 49(2/3):195–212
22. Karypis G, Kumar V (1995) METIS—Serial Graph Partitioning and Fill-reducing Matrix Ordering Library, Department Computer Science Engineering, University of Minnesota. http://www.cs.umn.edu/~karypis/metis
23. Karypis G, Kumar V (1998) Multilevel k-way partitioning scheme for irregular graphs. J Parallel Distrib Comput 48(1):96–129
24. Karypis G, Kumar V (1998) A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. J Parallel Distrib Comput 48(1):71–95
25. Hendrickson B, Devine K (2000) Dynamic load balancing in computational mechanics. Comput Methods Appl Mech Eng 184(2–4):485–500
26. Devine K, Boman E, Heaphy R, Hendrickson B, Vaughan C (2002) Zoltan data management services for parallel dynamic applications. Comput Sci Eng 4(2):90–97
27. Ozturan C (1995) Distributed environment and load balancing for adaptive unstructured meshes. PhD Thesis, Comput Sci Department, Rensselaer Polytechnic Institute
28. Ozturan C, de Cougny HL, Shephard MS, Flaherty JE (1994) Parallel adaptive mesh refinement and redistribution on distributed memory computers. Comp Methods Appl Mech Eng 119(1–2):123–127
29. Kirk BS, Peterson JW, Stogner RH, Carey GF (2006) libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations. Eng Comput 22(3):237–254
30. Stewart JR, Edwards HC (2004) A framework approach for developing parallel adaptive multiphysics applications. Finite Elem Anal Des 40(12):1599–1617
31. Remacle J-F, Shephard MS (2003) An algorithm oriented mesh database. Int J Numer Methods Eng 58(2):349–374
32. Wang S (2007) Krylov Subspace Methods for Topology Optimization on Adaptive Meshes. PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign
33. Wang S, de Sturler E, Paulino GH (2007) Large-scale topology optimization using preconditioned Krylov subspace methods with recycling. Int J Numer Methods Eng 69(12):2441–2468
34. Mangala S, Wilmarth T, Chakravorty S, Choudhury N, Kalé LV, Geubelle PH (2008) Parallel adaptive simulations of dynamic fracture events. Eng Comput 24(3):341–358
35. Huang C, Lawlor O, Kale LV (2003) Adaptive MPI. In: Proceedings of the 16th international workshop on languages and compilers for parallel computing (LCPC 2003). Lecture Notes in Computer Science, vol 2958, pp 306–322
36. The Message Passing Interface (MPI) standard library (2009) Argonne National Laboratory. http://www-unix.mcs.anl.gov/mpi
37. Choudhury N (2006) Parallel Incremental adaptivity for Unstructured Meshes in Two Dimensions. MSc. Dissertation, Department of Computer Science, Univesity of Illinois at Urbana-Champaign
38. Quinn MJ (2004) Parallel Programming in C with MPI and OpenMP. McGraw-Hill, New York
39. Kumar V, Gupta A (1994) Analyzing scalability of parallel algorithms and architectures. J Parallel Distrib Comput 22(3):379–391
40. Grama AY, Gupta A, Kumar V (1993) Isoefficiency: measuring the scalability of parallel algorithms and architectures. IEEE Parallel Distrib Technol 1(3):12–21