Comput. Methods Appl. Mech. Engrg. 266 (2013) 144–161

Contents lists available at SciVerse ScienceDirect

Comput. Methods Appl. Mech. Engrg.

journal homepage: www.elsevier.com/locate/cma

Scalable parallel dynamic fracture simulation using an extrinsic cohesive zone model

Rodrigo Espinha [a], Kyoungsoo Park [b], Glaucio H. Paulino [c,*], Waldemar Celes [a]

[a] Tecgraf/PUC-Rio Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rua Marques de Sao Vicente 225, Rio de Janeiro, RJ 22450-900, Brazil
[b] School of Civil & Environmental Engineering, Yonsei University, 50 Yonsei-ro, Seodaemun-gu, Seoul, Republic of Korea
[c] Department of Civil & Environmental Engineering, University of Illinois at Urbana-Champaign, 205 North Mathews Ave., Urbana, IL 61801, USA

A R T I C L E I N F O

Article history:
Received 23 July 2012
Received in revised form 12 July 2013
Accepted 15 July 2013
Available online 25 July 2013

Keywords:
Parallel topology based data structure
Parallel computing
Dynamic fracture
Extrinsic cohesive zone model

A B S T R A C T

In order to achieve realistic cohesive fracture simulation, a parallel computational framework is developed in conjunction with the parallel topology based data structure (ParTopS). Communications with remote partitions are performed by employing proxy nodes, proxy elements and ghost nodes, while synchronizations are identified on the basis of computational patterns (*at-node*, *at-element*, *nodes-to-element*, and *elements-to-node*). Several approaches to parallelize a serial code are discussed. An approach combining local computations and replicated computations with stable iterators is proposed, which is shown to be the most efficient one among the approaches discussed in this study. Furthermore, computational experiments demonstrate the scalability of the parallel dynamic fracture simulation framework for both 2D and 3D problems. The total execution time of a test problem remains nearly constant when the number of processors increases at the same rate as the number of elements.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Cohesive fracture simulations usually require a fine mesh discretization in order to capture non-linear behavior near the crack tip regions. This demands a large amount of computational resources, and thus realistic scale computations are generally limited. Alternatively, models of reduced geometries can be used in the simulations. However, reduced models do not accurately represent the original experiments, due to material-dependent length scales [1], and dependency of the direction of crack propagation on the mesh refinement level [1,2]. As real-scale cohesive fracture simulations may not be practical or feasible in a serial code executed on a single workstation, parallel processing becomes practically inevitable.

Current massively parallel environments are based on distributed memory architectures [14], in which each processor (or group of processors) of a computing node has private access to a region of the global system memory. Processors on different nodes communicate over a network by sending messages to each other in order to exchange data. In the context of finite element analysis, the model is divided into a set of partitions, which are distributed among the available processors such that computation can proceed in parallel. Network communication is necessary to synchronize simulation state data among mesh partitions.

Several parallel systems with support for distributed mesh representation have been proposed [3–9]. Some of them address various additional issues, like mesh adaptation and load balancing. Extrinsic cohesive fracture simulations, however, require that cohesive elements be adaptively inserted as cracks propagate. The parallel topological framework named ParTopS [10] provides support for the representation of fractured meshes and adaptive insertion of cohesive elements, for both 2D and 3D meshes of different types of elements. The framework is implemented as an extension of the serial TopS topological data structure [11–13].

In this paper, we propose an approach for the parallelization of three-dimensional dynamic fracture simulations based on ParTopS, which was presented in Ref. [10]. While Ref. [10] addresses the topological aspects of a distributed mesh representation required for parallel fracture simulations, and presents results of computational experiments that suggest the scalability of the topological framework, the current work builds on ParTopS to develop a compact programmer's interface for creating parallel numerical applications from an existing serial code. To this end, we extend ParTopS with a reduced set of collective functions that are inserted into the serial numerical code and require minor changes in order to enable parallel dynamic cohesive fracture simulation associated with microbranching and fragmentation. In addition, we discuss different strategies to parallelize an existing serial simulation and exploit replicated computations on different mesh partitions in order to minimize the need of network communication between pairs of partitions. Those strategies consider that an extrinsic

* Corresponding author. Tel.: +1 (217) 333 3817; fax: +1 (217) 265 8041.
E-mail addresses: paulino@illinois.edu, paulino@uiuc.edu (G.H. Paulino).

0045-7825/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.cma.2013.07.008

cohesive model of fracture be used, and thus cohesive elements are adaptively created and inserted along evolving crack paths, wherever and whenever determined by the analysis (there is no finite element analysis in Ref. [10]). The effectiveness and scalability of parallel simulations are evaluated through a set of computational experiments. We demonstrate the scalability of the topological framework on 1024 processors, which add and complement Ref. [10].

The organization of this paper is as follows. Related work is discussed in Section 2. Section 3 reviews TopS and ParTopS, the serial and parallel topological mesh representations used in this work to support parallel fracture simulations. In Section 4, we discuss the parallelization based on a sample simulation application by means of the introduction of the distributed topological mesh representation. Section 5 presents computational experiments that demonstrate the scalability of large simulations based on the proposed parallelization scheme. Concluding remarks and directions for future work are discussed in Section 6.

2. Related work

Support for parallel simulations has been one of the main features offered by recently developed systems for finite element analysis. Some of the important issues addressed by the current systems include: distributed mesh representation, parallel adaptivity and load balancing, interface to parallel solvers, and treatment of physical aspects of simulations. Several frameworks, such as PMDB [6,7], AOMD [4], FMDB [5], libMesh [8], ParFUM [3] and ParTopS [10], provide distributed representation of general unstructured meshes. The Zoltan library [27] also contains a set of utilities that are helpful for the management of distributed meshes, including mesh partitioning and communication procedures. Some systems, like the Sandia National Laboratory's SIERRA [9] framework and PETSc [28], address aspects related to both distributed meshes and the solution of partial differential equations. The existing parallel systems provide many fundamental services, which enable a wide range of finite element applications such as wave propagations [46,44], hydraulic fracture [45], fatigue crack growth [47], mantle convection simulations [48], etc.

Cohesive zone models of fracture introduce some challenges to the representation of distributed finite element meshes and to the simulation process [29]. When the intrinsic cohesive approach [30] is used, cohesive elements are created at inter-element interfaces before the fracture simulation starts. Since no mesh modification is required during the course of the simulation, parallelization becomes easier. The issues that must be handled by the mesh representation comprise the correct handling of mesh partitioning and inter-partition communications under the presence of cohesive elements. However, the cohesive elements act like nonlinear springs in the intrinsic approach, which may introduce significant artificial reduction of stiffness. On the other hand, in the extrinsic approach [20,22,23], cohesive elements are inserted (or activated) on demand when a fracture criterion specified by the simulation code is met. This makes a distributed mesh representation more complex, as topological changes may occur during the simulation, and thus must be handled locally and propagated to the neighboring partitions in a seamless and efficient manner. Serial algorithms for dynamic insertion of cohesive elements are currently available in the literature [31–33,13]. Due to the difficulties of the parallelization of the extrinsic cohesive approach, especially for three-dimensional meshes, few works have been presented to the best of the authors' knowledge.

Dooley et al. [34] employ a strategy based on element activation. In their approach, cohesive elements exist at every inter-element interface of the initial mesh, but remain inactive until the fracture criterion is met, when the traction-separation relation takes effect and the element becomes part of the simulation process. From a mesh topology point of view, this approach is equivalent to the intrinsic approach, in the sense that cohesive elements have been pre-inserted at all the inter-element interfaces at which cracks are expect to develop. The difference is that cohesive elements are made active to the simulation only when necessary. With the use of pre-inserted cohesive elements, the mesh topology does not need to change during the course of the simulation (unless adaptive mesh refinement techniques are employed). Hence, no communication is required in order to propagate topological changes to neighboring partitions, facilitating distributed mesh representation. On the other hand, it also introduces some issues that must be handled by the mesh representation framework. During the mesh construction, every node lying on inter-element interfaces is replicated as many times as the number of incident bulk elements, such that cohesive elements can be properly created. However, even when no cohesive element is currently active in the simulation, those elements, and the additional multiple node copies, still exist in the local mesh topology. This may introduce a significant overhead on the simulation, because the number of actual fractured facets is small if compared to the total number of facets in the model. Moreover, the actual mesh representation does not correspond to the one that is used for computation purposes. Therefore, the application itself or an additional access layer of the underlying mesh framework must be responsible for managing the consistent access to replicated nodes. While a cohesive element is not active, each pair of nodes shared between the two edges of a two-dimensional cohesive element corresponds to a single regular node in the simulation, although topologically represented as two different nodes. To overcome the inconsistency resulting from unnecessary nodal duplication and thus ensure mesh continuity in the simulation, a random node is chosen amongst the multiple node copies as the representative "root node". Nodal attributes are then computed and assigned to the root node. The difference between the mesh representation and the mesh used for simulation also affects the retrieval of some adjacency relationships between mesh entities. Access to those relationships do not have a natural and uniform treatment, since two bulk elements that should be adjacent to each other in the simulation are separated by an inactive cohesive element in the mesh representation.

The implementation of Dooley et al.'s approach [34] is based on the ParFUM framework [3], developed on top of the Charm++ parallel framework [16]. Hence, support is provided for the creation of "virtual processors" that are assigned to a smaller number of physical processors, which facilitate the overlapping of computation and communication in parallel applications. The parallel simulation code is split into two main routines: *init*() and *driver*() [34]. In *init*(), the whole unpartitioned mesh is loaded on a single processor. After *init*() has finished, ParFUM partitions the mesh and creates inter-partition communication infrastructure. The *driver*() routine runs in parallel, with one instance associated to each partition of the mesh, and executes the main time-step computations (explicit integration and synchronization of values on partition boundaries). Communication between neighboring partitions is done through a "ghost layer", consisting of read-only copies of elements and nodes from neighboring partitions, that is created around shared partition boundaries. Scalable dynamic fracture simulation results are presented for two-dimensional triangular meshes of up to 1.2 million elements and 512 processors. A complementary work by Mangala et al. [35] addresses parallel mesh adaptivity for two-dimensional meshes in ParFUM. In addition, ParFUM and the topology based data structure TopS have been integrated to investigate wave propagation in three-dimensional functionally graded media [44].

Radovitzky et al. [36] present an algorithm to parallelize extrinsic cohesive fracture simulations that is based on the combination of a discontinuous Galerkin (DG) formulation [37,38] for the continuum problem with cohesive zone models (CZM) to represent fractured faces [20,1,30,22,23]. Similarly to the intrinsic cohesive approach and Dooley et al.'s extrinsic approach [34], cohesive elements are created at every inter-element interface during the construction of the initial mesh. However, additional terms to the weak formulation of the problem, due to the introduction of the DG formulation, ensures that the simulation performs consistently in the absence of fracture, avoiding issues related to the intrinsic approach. The extrinsic cohesive law takes effect when the fracture criterion is met at a given inter-element interface, thus replacing the DG terms. Synchronization of nodal attributes of the simulation is done by adding local partition contribution to all the nodes in the partition and then completing boundary nodes by using an MPI [15] parallel reduced operation. Since cohesive elements are pre-inserted across the whole mesh domain, the requirement of propagation of topological changes is removed, and thus this approach shows to be scalable to a large number of processors. Results for wave propagation and dynamic fragmentation are presented for quadratic tetrahedral meshes (although, the procedure can be applied to other types of element) of up to 103 million elements on 4096 processors. Results for parallel simulation of three-dimensional crack growth for a spiral bevel pinion gear model have also been presented in Ref. [47].

The approach proposed in this paper for scalable three-dimensional fracture simulations is based on the distributed mesh representation provided by the ParTopS framework [10]. This framework is implemented as a parallel extension to the serial TopS [11,12] topological data structure, and to the algorithm for dynamic insertion of cohesive elements proposed by Paulino et al. [13]. Therefore, cohesive elements are allowed to be inserted into the mesh on demand, whenever the fracture criterion is met, as required by the extrinsic cohesive model formulation. The ParTopS framework ensures that no unnecessary node duplications occurs when cohesive elements are not present in the simulation and that topological changes are seamlessly and transparently propagated to neighboring partitions, in a scalable fashion [10]. As a consequence, global mesh topology is consistently maintained after every mesh modification operation, and the topological mesh corresponds to the simulation mesh. Cohesive elements are represented as regular mesh elements within the mesh topology, with uniform topological operators and algorithms applied to both two- and three-dimensional meshes, with possibly different types of elements. In order to reduce inter-partition communications, symmetrical mesh modifications and replicated computation of simulation data are exploited. Complex crack patterns are naturally handled without affecting scalability, which makes the framework suitable for micro-branching and fragmentation simulations.

3. Topological framework for fracture representation

In the following subsections the serial and parallel fracture mesh representations are briefly reviewed.

3.1. TopS: serial fracture representation

The adjacency-based topological data structure named TopS has been first proposed by Celes et al. [11] and further discussed in Refs. [12,13]. It provides a compact mesh representation (thus requiring a reduced storage space) for manifold meshes, while preserving the ability to retrieve all the adjacency relationships between any pair of mesh entities in time proportional to the number of retrieved entities. Of special interest here, it also provides a uniform interface for the representation and treatment of cohesive elements, with an efficient support for dynamic insertion of those elements, which is required for truly extrinsic fracture and fragmentation simulation. The concepts of TopS related to this work are briefly reviewed next.

3.1.1. Topological entities

The topological data structure defines various types of entities. However, only *element* and *node* entities are *explicitly* represented. This means that those entities have a concrete representation, which is actually stored in the memory space of the topological data structure. *Element* represents a finite element of any type that can be defined by templates of ordered nodes. This includes a wide range of the most commonly used finite elements (e.g. T3, T6, Q4, Q8, Tetra4, Tetra10, Brick20, etc.). *Node* represents finite element nodes, including corner and mid-side nodes. Along with element and node representations, the data structure also stores some adjacency relationships between those entities.

Other entities, namely *vertex*, *edge* and *facet*, are *implicitly* represented. Thus, their representations are created and retrieved "on-the-fly," whenever access to them is requested by the client application. *Vertex* represents an entity that is associated to a corner node; *edge* is a one-dimensional entity that is bounded by two vertices; and *facet* represents the interface between two elements, or an element and the mesh boundary. In 2D, a facet corresponds to a one-dimensional entity, whereas in 3D it corresponds to a two-dimensional entity. Therefore, the facet entity represents a convenient abstraction for a uniform treatment of operations acting on inter-element interfaces, such as the insertion of cohesive elements. In TopS, entities are retrieved and accessed by means of *opaque handles* returned by the topological data structure, rather than the entity itself. This also provides the client application with a transparent interface for the treatment of both explicit and implicit entities.

In addition to vertices, edges and facets, TopS defines another set of implicit entities that represent the *uses* of those entities by adjacent elements: *vertex-use*, *edge-use* and *facet-use*. The use of a facet by two elements is illustrated in Fig. 1. One facet-use is defined for each element that shares the facet. Each *entity-use* (vertex-use, edge-use or facet-use) is uniquely identified by the element (*Ei*) to which it is related and a local number (*id*) which identifies the entity within the fixed local element template: (*Ei*,*id*). Each vertex, edge or facet implicit entity is represented by one of its entity-uses. Analysis attributes can be attached to any mesh entity represented by TopS, either explicit or implicit.

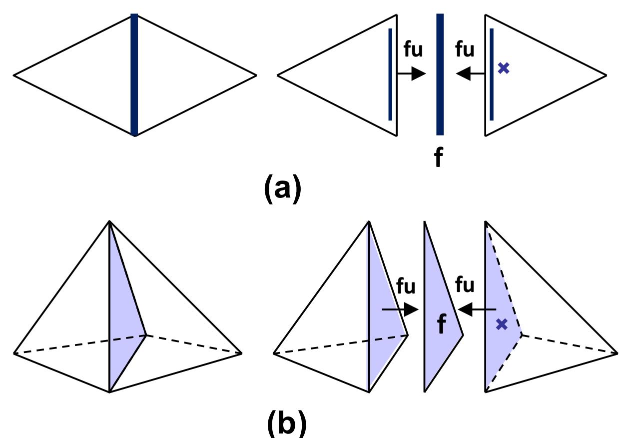

Fig. 1. The facet *f* is *used* by the two adjacent elements, in both 2D (a) and 3D (b) cases. One facet-use entity (*fu*) is defined for each element that uses the facet. The facet is represented by one of its uses (indicated by the 'x' mark).

3.1.2. Dynamic insertion of cohesive elements

To the extent of the topological mesh representation, cohesive elements are types of elements that consist of two facets. They can be inserted at the interfaces between two adjacent elements in the mesh (internal facets) in order to provide a representation of fractured facets. Cohesive elements may contain nodes that are shared by their two facets. Fig. 2 shows two cohesive elements (CohE3 and CohT6) that match the facets of two bulk elements (T6 and Tet10, respectively). In TopS, cohesive elements are explicitly represented and treated like any other regular element.

Dynamic insertion of cohesive elements follows the systematic classification of topological facets proposed by Paulino et al. [13]. This scheme provides a consistent classification that can be uniformly employed for both 2D and 3D meshes. For a detailed description of the algorithm for inserting a cohesive element into a mesh we refer the reader to Ref. [13].

3.2. ParTopS: parallel fracture representation

A topological distributed mesh representation framework with support for adaptive insertion of cohesive elements was proposed in Ref. [10]. The framework, named ParTopS, is implemented as an extension to the serial TopS topological data structure [11–13]. The ParTopS framework, briefly reviewed in this section, will be used as the topological support for the proposed parallel fracture simulation scheme.

3.2.1. Distributed mesh representation

The distributed mesh representation consists of disjoint partitions of the set of entities that make up the finite element mesh. Each partition is assigned to exactly one (logical) processor, and comprises a regular serial TopS mesh, with an additional *communication layer* (see Fig. 3). The communication layer is composed of local copies of entities from remote partitions. It is constructed around the nodes of the partition boundaries, and provides local access to data from neighboring partitions that are necessary for local computations.

In addition to providing access to remote entities, the communication layer is also utilized by the ParTopS framework in order to facilitate the updating of mesh topology when cohesive elements are concurrently inserted into multiple partitions. To this end, two types of entities are defined: *proxy* and *ghost*, which are shown in Fig. 3. *Proxy* nodes and elements are exact local copies of the corresponding entities from remote partitions; they are treated like any other regular local entity and can be accessed or edited, and can store analysis attributes. *Ghost* nodes are also local copies of remote entities; however, unlike proxies, their direct adjacent entities are not represented. Any implicit entity (vertex, edge or facet – and the corresponding *entity-uses*) is considered as a ghost if all of its incident nodes are ghosts. Otherwise, either it is considered as a local or proxy entity, following the type of one adjacent *representative* element. The ParTopS framework uses ghosts to define the boundaries of the communication layer, and to ensure local topological consistency of each partition's sub-mesh. The separation of entities of the communication layer into proxies and ghosts is central to parallel adaptive insertion of cohesive elements, which uses it in order to treat ambiguities that arise when cohesive elements are inserted across partition boundaries.

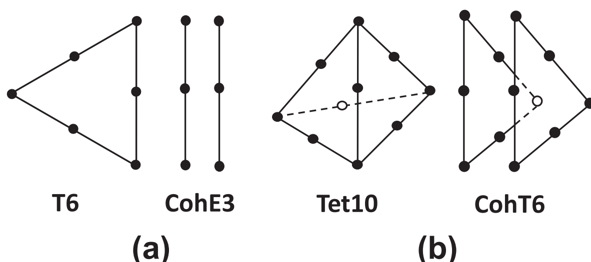

Fig. 2. Examples of 2D (a) and 3D (b) bulk elements and the corresponding cohesive elements.

Each local or proxy entity has its complete set of directly adjacent entities represented in the local mesh partition. Hence, computations that depend on data from adjacent entities can be performed locally, without the need for communication with remote partitions, as long as the required data are up-to-date. This makes the definition of proxy entities as editable entities convenient to any situation when it is possible to reduce the amount of inter-partition data synchronization by employing replicated local computations, as exploited by the proposed parallel simulation framework. Ghost entities, on the other hand, have an incomplete local adjacency set, which makes computation at these entities dependent on remote adjacent information. Therefore, most data at ghost entities are updated from results of computations performed at partitions containing a local or proxy representation of the corresponding entities.

In ParTopS, each entity is owned by a single partition; the owner partition manages the so-called *real entity* and its attributes, while other partitions may have proxy or ghost representations of it. Each proxy or ghost entity has a reference to the corresponding real entity, which is defined by the unique tuple (*owner_partition*, *owner_handle*), where *owner_partition* is the *id* of the partition that owns the entity, and *owner_handle* is the local handle (opaque identifier) of the entity in that partition.

3.2.2. Parallel insertion of cohesive elements

A topology-based algorithm for parallel adaptive insertion of cohesive elements is also presented in Ref. [10]. The algorithm extends the serial version of Ref. [13] and works for both 2D and 3D meshes with different types of elements. The so called symmetrical topological operators are defined here. These operators produce the same results in every partition that concurrently modifies a mesh entity. They are employed in order to avoid the use of exclusive access locks to entities and thus reduce inter-partition communication to update mesh topology.

The application is responsible for the identification of newly fractured facets, at which cohesive elements will be adaptively inserted. The list of fractured facets of each partition includes both local and proxy entities, and must be consistent with respect to all the partitions sharing a copy of the same facet. This means that if a facet is fractured in a partition, then it is fractured in every partition with a copy of it; otherwise it is not fractured in any partition.

The parallel algorithm for inserting cohesive elements consists of three phases. In the first phase, elements are inserted at fractured facets of each partition using the serial algorithm of Ref. [13]. Two constraints are included so that topological results obtained are the same for every partition, regardless of the order in which cohesive elements are inserted in each of them. At the end of the first step, topology of local and proxy entities is consistent; however references of newly created proxy entities (cohesive elements and duplicated nodes) to the corresponding real entities need to be updated. Ghost entities also need to be updated.

The second phase computes the missing references of the newly created proxy entities, which consist of the identifiers of their real entities. Therefore, requests are sent to the neighboring partition that owns each proxy entity. The remote partition then looks up the entity corresponding to that proxy entity and sends the identifier of the corresponding real entity back to the local partition, which updates its proxy reference.

In the third phase, ghost nodes affected by nodal duplications in remote partitions are updated. They either have their references to

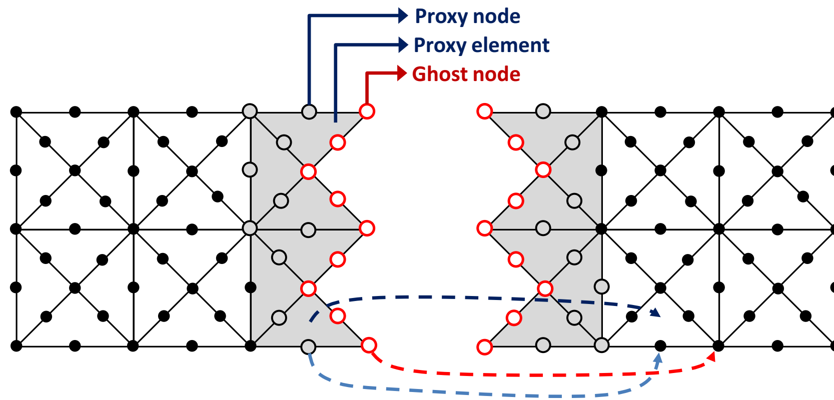

Fig. 3. A sample 2D mesh divided into two partitions. A communication layer is added to partition boundaries. Proxy and ghost entities are indicated in the figure. Each of them has a reference to the corresponding real entity represented in a remote partition.

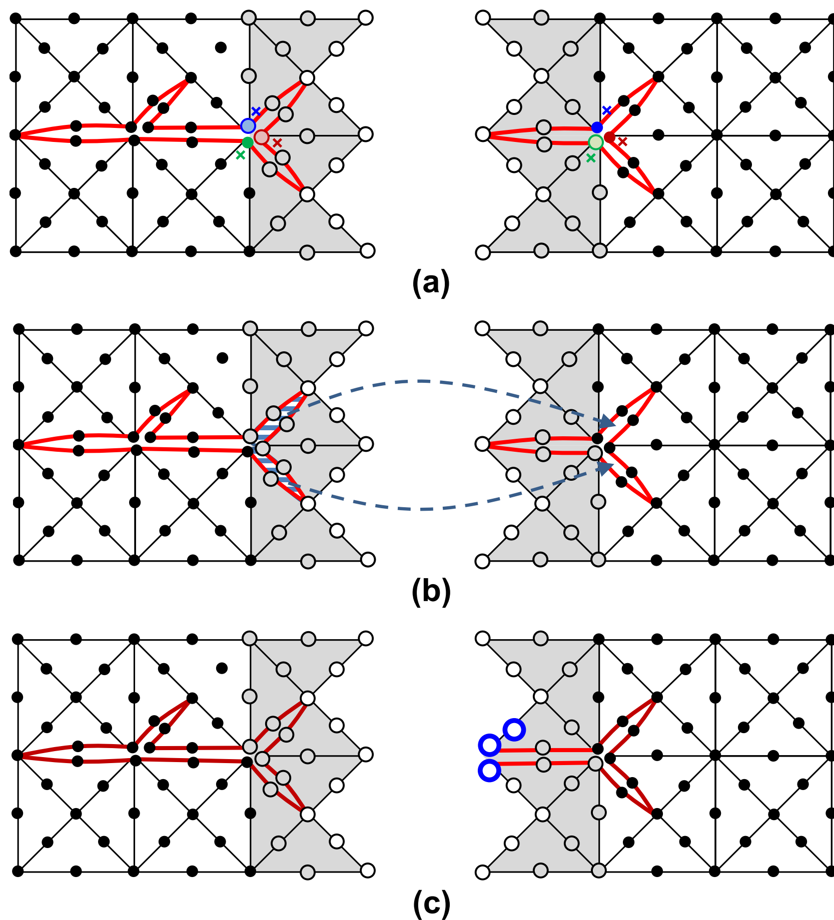

Fig. 4. The three phases of the parallel insertion of cohesive elements. (a) After phase 1, topology of local and proxy entities is consistent, regardless of the order of insertion of cohesive elements in each partition. (b) Phase 2 updates references from proxy elements and nodes to the corresponding real entities. (c) After phase 3, the topology of ghost nodes affected by nodal duplications is updated.

real entities replaced or split into two or more new (ghost) nodes in order to maintain mesh topology consistent. The three phases of the algorithm are illustrated in Fig. 4.

4. Parallel dynamic fracture simulations

One of the goals of the current work is to provide a simple and compact parallel programmer's interface such that an existing serial code can be efficiently parallelized with a minimum of additional effort, through reusing an existing fracture simulation code.

In this section, we describe the parallelization of dynamic fracture simulation by means of a distributed topological representation of the mesh. Our proposal is first presented based on the parallelization of an existing serial application for Mode I fracture. The solution is then extended to microbranching simulations. The application code, originally implemented on top of the TopS mesh representation [11–13], is parallelized by the introduction of a small set of parallel function calls provided by the proposed topological framework based in ParTopS [10]. Minor changes are required to the

application in order to execute it in a distributed environment.

The *Application Programming Interface* (*API*), exported to client applications, follows the traditional *Single Program Multiple Data* (*SPMD*) paradigm [14], in which the same program executes concurrently for each partition of a data set. The API uses message passing for inter-partition communication, and can be integrated into an existing code based on the *MPI* [15] message passing standard. Internal implementation is done on top of Charm++ [16,17], an object-oriented framework for parallel applications, which is based on asynchronous message passing. However, although asynchronous message passing paradigm possesses great potential for performance improvements (through overlapping of communication and computation), it tends to increase code complexity considerably. Hence, considering the intrinsic complexity of physics-based numerical simulations, we instead provide a possibly less efficient, but simple, easy-to-use functional interface. The proposed distributed topological framework consists in a set of regular serial functions for manipulating and accessing the local mesh partition, plus a reduced set of parallel *collective* functions that encapsulate inter-partition communications. The collective functions are concurrently called by every mesh partition. Thus, we aim to keep the application coding as simple as possible, while still achieving effective and scalable systems.

4.1. Synchronization requirements

In dynamic finite element analyses, the computation of a current time step depends on the results of a previous time step. Likewise, intermediate computations within a time step also depend on previously computed data. Data dependency limits the capacity of parallelization, as dependent computations cannot be executed concurrently. Hence, *synchronization points* are introduced into the simulation whenever remote data required by a computation are not available to the current mesh partition, which must wait until the data are received from its neighbors. As a result, opportunities for parallelization lay in the decomposition of the geometric domain into separate pieces that can be processed independently in between two consecutive synchronization points. For example, when nodal stresses are computed in a time step, displacements have been previously computed. Once displacements are up-to-date, stresses can be computed concurrently by the mesh partitions.

Every mesh partition corresponds to a sequential unit of computation. In each partition, numerical computations are performed mostly based on local data. However, simulation attributes associated to topological entities located on boundaries of neighboring partitions must be maintained consistent among them.

The ParTopS communication layer provides local access to adjacent entities located at other partitions, as required by computations at entities on partition boundaries. In order to maintain data consistent among mesh partitions, four approaches are evaluated next. The first approach corresponds to the conventional strategy, in which entities in the communication layer are considered as read-only, and results of computations are stored only at local entities of a mesh partition. Since attributes of an entity are exclusively updated by a single partition (the entity's owner), race conditions are avoided – this corresponds to an implicit *exclusive access locking* mechanism [18]. However, entities in the communication layer must be synchronized with the corresponding remote real entities whenever the entities are modified and are required by local computations.

The other three approaches exploit the properties of the ParTopS communication layer, in order to reduce the number of synchronization points required within a simulation time step. Hence, entities in the communication layer are editable, and thus results of computations are stored at them whenever it is convenient for the client application. However, as data associated to a topological entity can be modified concurrently by several different mesh partitions, we need to ensure that this happens seamlessly. To this end, we propose the use of *replicated computations* among mesh partitions, in a similar manner as the topological operators employed in the parallel insertion of cohesive elements (Section 3.2.2). As a result, the number of synchronization points required by the simulation can be reduced. The concept of symmetry of computations is discussed in Section 4.4.

For an analysis application, computational results are stored as *attributes* associated to mesh entities. At each synchronization point, attributes of topological entities in a mesh partition are *synchronized* with respect to the other partitions, in order to keep them updated for computations that follow. Note, however, that not every attribute of a topological entity must be synchronized at each synchronization point, but only those that require consistency for the next computation. Synchronization can be postponed until data are actually needed, or performed asynchronously while other independent computations are running. Nonetheless, the application must block execution if the required data are not available at the expected time.

4.2. Structure of a serial fracture simulation

The basic algorithm of a serial numerical fracture simulation [19,42] is summarized in Algorithm 1. The central differences method (i.e. explicit method) [21] is utilized for time integration, and the *Park–Paulino–Roesler* (*PPR*) constitutive model [20] is used for the traction-separation relation of the extrinsic cohesive zone model [22,23,20]. The structure of the serial simulation shown in Algorithm 1 is used as a basis for the discussion of our proposal for parallel simulation.

Algorithm 1. Basic structure of a serial dynamic fracture simulation.

```
Initialize the finite element model

For each time step
  1 - Compute current applied boundary conditions
  2 - Check the insertion of new cohesive elements
    2.1 - Compute nodal stresses
    2.2 - Check fracture criteria along facets (e.g.
          a stress-based criterion)
    2.3 - Insert cohesive elements
    2.4 - Update nodal masses
  3 - Compute the internal force vector
  4 - Compute the cohesive force vector
  5 - Update nodal accelerations and velocities
  6 - Update external forces
  7 - Apply boundary conditions to nodes
  8 - Update nodal displacements
```

Simulation starts just after the initialization of the finite element model. At each time step, fractured facets must be identified, and cohesive elements are inserted along them (item 2). In this study, the fracture criteria is based on nodal stresses, which must have been previously computed (item 2.1). Stresses are first calculated at the Gauss points of each volumetric element, from the attributes of the element's nodes. Then stresses at the Gauss points are extrapolated to the nodes using shape functions. The resulting stresses at a node are obtained by averaging contributions of all the adjacent volumetric elements. Once stresses are up-to-date, internal facets (interfaces between volumetric elements) are checked

against a fracture criterion (item 2.2), and the ones for which the criterion is met are marked as *fractured* in the current step. If there are facets marked as *fractured*, cohesive elements are created *on demand* and inserted along the fractured facets (item 2.3). The attributes of duplicated nodes during the insertion of cohesive elements are copied from the original nodes inside *callback functions* registered by the application and called by TopS after each nodal duplication. The application is also notified of each cohesive element inserted by a callback function, which is responsible for setting up the element. Then, the attributes of the new cohesive elements are calculated. In addition, when cohesive elements are inserted, nodal masses are updated by summing up the contributions of adjacent volumetric elements (item 2.4).

After checking the insertion of cohesive elements (item 2), the internal nodal forces (item 3) are computed by summing up the contributions of the forces of the volumetric elements adjacent to the nodes. Element contribution is based on element stiffness and displacements of adjacent nodes. Then, the cohesive nodal forces (item 4) are computed for the nodes adjacent to cohesive elements. First, cohesive separations at the integration points are calculated from the nodal displacements. Next, the cohesive tractions and the corresponding cohesive nodal forces are computed in conjunction with calculated separations and other element attributes (e.g. material properties and history information associated with the traction-separation relationship). Then, the resulting element contribution is added to adjacent nodes. Finally (items 6 to 8), the remaining nodal attributes (accelerations, velocities, external forces, boundary conditions and nodal displacements) are updated based on attributes of the nodes themselves.

Note that the nodal displacements are generally computed at the beginning of each time step for the explicit time integration [19,42]. However, in this study, the nodal displacements are updated at the end of each time step in order to reduce the number of synchronizations in the parallel simulation. Thus, computed stresses, internal forces, and cohesive forces are based on the nodal displacement computed in the previous time step. In other words, the nodal displacement computed at the end of the current time step (n) is the nodal displacement field for the next time step ($n+1$).

4.3. Computation patterns

Before discussing the parallelization of fracture simulation, we note that our basic types of computation patterns, regarding the origin and destination of data associated to topological entities, can be identified in regular finite element computations. We analyze each one of these patterns in light of parallelization. The identified computation patterns are here named: *at-node*, *at-element*, *nodes-to-element*, and *elements-to-node*. Computations of types *at-node* (Fig. 5(a)) and *at-element* (Fig. 5(b)) are self-contained at the corresponding entities, thus depending only on data associated to the entities which are modified. For example, in the central difference method, current nodal displacements are computed on the basis of previous nodal information (e.g. displacement, velocity and acceleration), which corresponds to the *at-node* pattern. For the cohesive traction computation, separation history information is required, and thus associated with the *at-element* pattern. These computations can be naturally performed in parallel. In the *nodes-to-element* pattern, element results are computed from the contributions of the nodes adjacent to the element (Fig. 5(c)). For instance, the stress computation at the Gauss points corresponds to the *nodes-to-element* pattern. Once the input nodal attributes are consistent, the computation can happen locally for the target element. There is no writing conflict since each element is assigned its corresponding resulted value. On the other hand, in the *elements-to-node* pattern (Fig. 5(d)), element contributions are added to the results stored at adjacent nodes. The nodal stress computation is an example of the *elements-to-node* pattern. When an appropriate topological data structure is employed, the nodal results can be computed consistently by traversing over the list of nodes in the finite element mesh. For each node, the adjacent elements are visited and their contributions computed and added to the node. However, this pattern is commonly, and more efficiently, implemented by traversing over the list of elements in the mesh. Then, the contribution of each element is computed and added to the adjacent nodes based on the element connectivity information. With this procedure, writing conflicts do arise, since different elements contribute to the value stored at a single node.

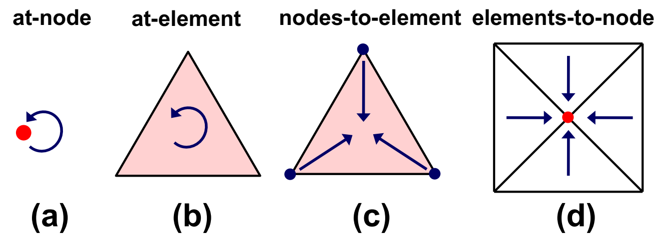

Fig. 5. The four basic computation patterns found in regular finite element analyses. (a) *at-node*: the result computed at a node depends only on the node itself; (b) *at-element*: the result of an element depends only on the element itself; (c) *nodes-to-element*: the result at an element depends on the adjacent nodes; (d) *elements-to-node*: the result at a node depends on the contributions of adjacent elements.

Complicated computations can be decomposed into smaller computations that conform to the four patterns described above. The classification of computations into a set of more basic patterns provides a systematic and abstract way such that required synchronization points can be easily identified in the numerical simulation code. This is based solely on the computation structure, regardless of the understanding of the details of the numerical simulation.

4.4. Replicated symmetrical computation and stable iterators

For the purpose of the current work, a computation performed in parallel is defined as *symmetric* if it produces the same results in all partitions. Thus, using symmetric computation, we are able to process local and associated proxy topological entities in parallel, with no need for inter-partition communication to synchronize the computed data. Therefore, we follow the same philosophy as the symmetrical topological operations employed in the parallel insertion of cohesive elements (Section 3.2.2), in which communication is replaced by replicated computations at the topological entities (local, proxy or ghost) of every mesh partition.

However, floating-point operations (e.g. addition and multiplication) are not associative [39], and thus different results can be obtained due to the order in which operations are performed. Hence, computations concurrently executed by different partitions for the same topological entity can produce diverging results if the order of floating-point operations is not identical for all of them. Numerical differences, even small, can make binary decisions (e.g. whether a facet has met the fracture criterion) to behave differently in each partition. In addition, the differences can accumulate along a large number of time steps, affecting computational results.

If a consistent ordering of floating-point operations can be ensured, then identical (*symmetrical*) results can be achieved for a topological entity in every mesh partition, assuming that all processors use the same floating-point number representation. To this end, we first observe the symmetrical characteristics of the computation patterns discussed in the previous section. Computations of

types *at-node* and *at-element* are naturally order-independent, and thus symmetric, since only local access to the modified entity is required. In the *nodes-to-element* pattern, the results of an element depend on the order in which contributions of adjacent nodes are combined. However, this nodal ordering is imposed by the fixed local topology of the element, which is consistent for every mesh partition owing a copy of the element. Therefore, nodes-to-element computations are also naturally symmetric. On the other hand, in *elements-to-node* computations, the results of a node depend on the order in which contributions of adjacent elements are combined.

As discussed in Section 4.3, serial *elements-to-node* computations are traditionally performed by traversing the list of elements and adding the contribution of each element to its nodes. In order to parallelize these computations in a symmetrical way (with identical results obtained for every mesh partition), the elements must be visited according to a globally consistent ordering in the distributed mesh. To this end, we employ an approach that is based on an *implicit* global ordering of elements. *Stable iterators* for element traversal are included into the distributed mesh representation of ParTopS such that ordered element traversals can be transparent to the client application.

Implicit global element ordering is achieved by lexicographic comparison of the tuples (*owner_partition, owner_element_handle*), which uniquely identify any element in the distributed mesh. Thus, elements are compared first by owner partition (*owner_partition*) and, if they are in the same partition, by the local element handles with respect to that partition (*owner_element_handle*). No explicit global identifiers need to be assigned to elements, and corresponding maintenance costs are avoided. In addition, inter-partition communication in order to update the global ordering when the mesh is modified is not required. This approach differs from others [34], which employ 128-bit precision operators in order to ensure the consistency of double precision (64-bit) computations at nodes shared by neighboring mesh partitions.

At each partition, ParTopS stores two distinct arrays of elements: the first array stores the local elements, and the second stores the proxy elements. In order to implement a stable element iterator, the proxy-element array is kept ordered by an element global identifier (the tuple [*owner_partition*, *owner_element_handle*]). The local-element array is implicitly ordered, as element handles correspond directly to locations in the array. The stable iterator then traverses the elements in the partition in the following order: each element in the proxy-element array is visited as long as its corresponding owner partition identifier is less than the identifier of the current partition. Then, the local-element array is traversed and, finally, the proxy-element array traversal is resumed.

Different ordered arrays and stable iterators can be created for each distinct group of elements. In our fracture simulations, separate iterators were used for volumetric and cohesive elements. The main advantage of this is that the volumetric array does not need to be reordered whenever cohesive elements are inserted into the mesh in this study. This is because *elements-to-node* computations of volumetric elements are performed separately from the computations of cohesive elements. As the number of cohesive elements is usually much smaller than the number of volumetric elements, the reordering effort during the fracture simulations tends to be negligible.

Based on distributed mesh representation provided by ParTopS, symmetrical computations can be done for both local and proxy entities for all identified computation patterns. Ghost entities are synchronized with results computed at the corresponding remote entities (of the local or proxy type). Only the pattern *at-node* can be performed symmetrically on ghost nodes.

4.5. Distributed attribute representation

The TopS serial mesh representation allows a client application to attach analysis attributes to any type of topological entity in a model, including implicitly represented entities (i.e. facets, edges and vertices). The attributes are linked to the unique identifiers of the topological entities to which they are attached, and stored as generic pointers to the memory locations where the actual data can be found. The identifier of an implicit entity is related to the adjacent element that is used to represent the entity, and so this may change if elements are removed from the mesh. However, the data structure is capable of handling such cases, which are discussed in Ref. [12]. For the purpose of the current discussion, though, we assume that no element will be removed.

In order to facilitate attribute management by the client application, the original serial attribute representation was extended with *attribute sets* that are allocated and managed by the topological framework. Two types of sets are defined: *dense* and *sparse*. *Dense* attributes are optimized for representing data attached to all (or almost all) the entities of a given type. This includes the simulation state stored at every node and element of the model. *Sparse* attributes, on the other hand, are intended for representation of data associated to smaller sets of entities; an example is boundary conditions associated to boundary nodes.

Dense attributes may be attached to explicit entities (nodes and elements); only one different set is created for each type of element (e.g. T3, TET4, T3_PROXY, TET4_PROXY) or node (e.g. LOCAL, PROXY, GHOST). Implementation consists of generic dynamic arrays indexed by entity identifiers, constrained to the entity type to which each array corresponds. Attribute arrays are automatically resized by the topological framework whenever elements or nodes are inserted into or removed from the mesh. Several sets can be attached to the same type of entity, and each set has a unique identifier, which is used by the application in order to refer to it. A callback function for packing/unpacking attributes into/from a stream of bytes is supplied by the application so that attributes can be serialized and transmitted over a network. Since array allocation is managed by the topological framework, the application must provide the attribute storage size during array creation. An attribute may be either a pointer to a memory location or the actual data representation. The application is responsible to fill in each position of the array with an appropriate attribute to the corresponding entity; a default initialization function may also be provided by the application in order to set the initial attribute values.

Sparse attributes can be attached to any type of entity, either explicit or implicit. Like dense attributes, different sets are created for each type of element, node or implicit entity. Implementation consists of ordinary dynamic associative hash tables indexed by entity identifiers.

Attribute sets are implemented at the level of the TopS serial data structure. Construction of distributed sets is done by means of an additional collective function, called at every processor simultaneously. The collective function delegates to local serial functions at the mesh partitions in order to create local attribute sets for the requested entity type. The same unique global identifier is returned by every partition, so it can be passed around without the need for explicit mapping. The attribute sets allow one to write a serial or parallel numerical simulation code under the same common interface, with a few extensions for parallel computations. These extensions consist of support for proxy and ghost entities and collective functions for synchronizing attributes of such entities. In order to synchronize an attribute set attached to ghost or proxy entities, a collective function `SyncAttrib(entity_type, attrib_id)`, which parameters are the type of the entity to which the attribute is attached (`entity_type`) and the global attribute identifier with

respect to that type (`attrib_id`), is called at every partition. The current partition determines its neighbors that own such entities and the local handles of the elements at them. Messages requesting the attribute's data are sent to all the neighboring partitions, which then access the local elements and reply with the requested values.

4.6. Parallel functions exported to the numerical application

A brief description of the main set of functions of the parallel Application Programming Interface (API) provided by the proposed topological framework is presented:

```
void topParInit();
TopParModel topParModel_Create();
int topParModel_ReadMesh(TopParModel model,
  char* format, char* meshfile, char* partfile);
TopModel* topParModel_GetLocalMesh(
  TopParModel model);
TopAttribId topParModel_CreateNodeDenseAttrib(
  TopParModel model, size_t sizeof_attrib);
void* topModel_GetNodeDenseAttribAt(
  TopModel* mesh, TopAttribId attribid,
  TopNode node);
void topParModel_SyncProxyNodeDenseAttrib(
  TopParModel model, TopAttribId attribid);
void topParModel_SyncGhostNodeDenseAttrib(
  TopParModel model, TopAttribId attribid);
void topParModel_SyncFacets(
  TopParModel model, TopFacet* facets);
void topParModel_InsertCohesiveAtFacets(
  TopParModel model, ElemType type, TopFacet*
  facets);
```

The collective function `topParInit`, which is concurrently called by every mesh partition at the beginning of the application execution, initializes the support for the distributed mesh representation. The collective function `Create` (the *topParModel* prefix is omitted hereafter) creates a new empty distributed finite element model and returns the identifier of the created model (`TopParModel`). The utility collective function `ReadMesh` loads a distributed mesh from a pair of input files (`meshfile` and `partfile`, containing the original mesh and mesh partition description, respectively) in a given file format (`format`) into the distributed model (`model`). The local function `GetLocalMesh` returns a pointer to the local mesh of the current partition, represented by the serial TopS topological data structure along with the parallel extensions that make up ParTopS. The collective function `CreateNodeDenseAttrib` creates a new attribute set that is associated to every node in the distributed mesh. The function returns the identifier of the created attribute set (`TopAttribId`), which is the same for every mesh partition, and can be used in order to access the corresponding attribute of a node by calling the local function `GetNodeDenseAttribAt`. The function returns a pointer (`void*`) to the memory address corresponding to the node's (`node`) data in the local mesh (`mesh`). The functions `SyncProxyNodeDenseAttrib` and `SyncGhostNodeDenseAttrib` synchronize the attributes of proxy and ghost nodes, respectively, with regard to an attribute (`attribid`) associated to the corresponding remote nodes. The collective function `SyncFacets` synchronizes local lists of fractured facets (`facets`) in each mesh partition with their neighboring partitions. Finally, the interface to the parallel adaptive insertion of cohesive elements is provided by the collective function `InsertCohesiveAtFacets`, which receives as parameters the type of the elements to be inserted (`type`) and the list of fractured facets of the current partition (`facets`), including both local and proxy facets. In order to parallelize the insertion of cohesive elements, the application replaces the serial function with the parallel version, which has a similar signature. The collective call does not return the list of inserted cohesive elements; instead the application is notified for each element inserted and duplicated nodes (local, proxy or ghost) through registered callback functions. The callback interface is mostly the same for both the serial and parallel implementations. Upon notification, the application can set the attributes of the new nodes and elements.

4.7. Structure of the parallel simulation

The structure of the parallel simulation is based on the synchronization requirements and the computation patterns discussed in the previous sections. In a general sense, the parallel code corresponds to the original serial code concurrently executed on every mesh partition, and with a few additional parallel collective functions for the synchronization among mesh partitions. The parallel numerical application is described next, considering four different approaches for synchronization, as first mentioned in Section 4.1.

4.7.1. Parallel approach based on computations at local entities only

In the parallel simulation structure shown in Algorithm 2, the results of concurrent numerical computations are stored only at local entities in each mesh partition. This corresponds to the first approach discussed in Section 4.1, and is equivalent to the conventional strategy based on the implicit use of exclusive access locks at mesh entities. Entities in the communication layer (proxies and ghosts) are considered as read-only entities, thus serving to the only purpose of providing the data required by local computations near partition boundaries. Data consistency of proxy and ghost entities is ensured by attribute synchronization with neighboring partitions. Synchronization points are emphasized in bold in Algorithm 2.

Algorithm 2. Basic structure of the parallel fracture simulation, considering an approach based on computations on local entities only.

```
Initialize the finite element model

For each time step
  1 - Compute current applied boundary conditions
  2 - Check the insertion of new cohesive elements
    2.1 - Compute nodal stresses
    A1. Synchronize attributes of proxy nodes
    2.2 - Check fracture criteria along facets
    A2. Synchronize local sets of fractured facets
    2.3 - Insert cohesive elements
    2.4 - Update nodal masses
  3 - Compute the internal force vector
  4 - Compute the cohesive force vector
  A3. Synchronize attributes of proxy cohesive
      elements
  5 - Update nodal accelerations and velocities
  6 - Update external forces
  7 - Apply boundary conditions to nodes
  8 - Update nodal displacements
  A4. Synchronize attributes of proxy nodes
  A5. Synchronize attributes of ghost nodes
```

At the beginning of each time step, it is assumed that all the partitions of the finite element model are consistent. During time step execution, however, attributes of nodes and elements are altered by intermediate computations, which happen independently on each partition. Those attributes must be turned consistent in order to be used by subsequent computations. The synchronization requirements for maintaining data consistency among mesh partitions are analyzed next, based on the basic structure of the serial numerical simulation described in Section 4.2.

Computation of nodal stresses (item 2.1) consists of two sequential stages. First, for each volumetric element, the attributes of adjacent nodes are temporarily copied to the element, and stresses are calculated at the Gauss points within an element. Next, the nodal stresses are extrapolated from the Gauss points and are added to the adjacent nodes in the current partition, but only local nodes are updated. Note, however, that a local node can be adjacent to both local and proxy elements; thus contributions corresponding to those elements must be computed. As a proxy element can be adjacent to either local, proxy or ghost nodes, the attributes of those entities must be consistent prior to the beginning of the stress computation. This is ensured by the synchronization of proxy and ghost nodes that takes place at the end of the previous time step (items A4 and A5), just after the computation of nodal displacements.

Fracture determination (item 2.2) has to be consistently done for every local facet of the current mesh partition. However, local facets may have incident proxy nodes. For that reason, proxy nodes must be synchronized (item A1) before fracture criterion evaluation. Insertion of cohesive elements, when needed, is performed by a parallel collective function with well-defined inputs and outputs (Section 4.6), which requires that proxy facets are provided in addition to the local ones. Therefore, the local set of fractured facets of the current partition is synchronized with neighboring partitions (item A2). After inserting cohesive elements (item 2.3), the global mesh topology and attributes that correspond to the affected entities are consistent.

If cohesive elements are inserted, nodal masses (item 2.4) are updated from the contributions of adjacent volumetric elements. The element mass matrix does not change during the simulation, and subsequent computations in a time step do not require, as input, the masses of either proxy or ghost nodes. Hence, synchronization of nodal attributes is not needed either before or after this computation.

The internal force vector (item 3) is computed in two stages, in a similar way as the nodal stresses. For each volumetric element, attributes of its adjacent nodes are temporarily copied to the element in order to be combined with other additional element attributes. Then, the element's contribution is added to the adjacent local nodes. Displacements are the only nodal attributes used in this computation. As they have not been altered in the current time step to this point, synchronization of proxy or ghost entities is not required in order to compute internal forces.

Computation of the cohesive forces (item 4) can be decomposed into three separate parts. In the first one, for each cohesive element, temporary attributes (e.g. cohesive separations) are computed at the element from adjacent nodal data (e.g. displacements). In the second part, element attributes (e.g. history information) are updated and combined with other temporary attributes, and the results (e.g. cohesive tractions) are stored back at local elements. Finally, the element force contribution is added to each adjacent node. This computation affects both local cohesive elements and local nodes adjacent to them. Since local nodes can be adjacent to either local or proxy elements, and those elements, in turn, to either local, proxy or ghost nodes, computation will depend on the attributes associated to those entities. However, as input nodal attributes have not changed to this point in the current time step, we only need to synchronize the attributes of cohesive elements (item A3), such that they are up-to-date for the next time step.

The remaining computations (items 5 to 8) depend only on the attributes of the nodes that they modify. Thus, they are self-contained and do not depend on synchronization with neighboring partitions. At the end of the time step, attributes of proxy and ghost entities are synchronized (items A4 and A5) such that they are consistent for the next time step.

Because element and node attributes are modified by a single mesh partition (the one which owns the entity), numerical divergences between neighboring partitions are naturally avoided. Conversely, attributes of proxy and ghost entities must be synchronized accordingly whenever the entities are modified.

4.7.2. Parallel approach based on replicated computations

In this section, replication of numerical computations at elements and nodes in the communication layer of each mesh partition is utilized to reduce the number of attribute synchronization points needed. In this scenario, attributes of topological entities in the communication layer can be edited and results of computations are stored at them whenever it is convenient for the numerical application. Next, the structure of the parallel simulation of Section 4.7.1 is revisited (see Algorithm 3), and the computation patterns found are analyzed.

Algorithm 3. Basic structure of the parallel numerical simulation based on replicated computations. Attributes of proxy nodes are sporadically synchronized, at fixed intervals of n time steps.

```
Initialize the finite element model
For each time step
  1 - Compute current applied boundary conditions
  2 - Check the insertion of new cohesive elements
    2.1 - Compute nodal stresses
    If the current time step is multiple of n
      A1. Synchronize attributes of proxy nodes
    2.2 - Check fracture criteria along facets
    A2. Synchronize local sets of fractured facets
    2.3 - Insert cohesive elements
    2.4 - Update nodal masses
  3 - Compute the internal force vector
  4 - Compute the cohesive force vector
  If the current time step is multiple of n
    A3. Synchronize attributes of proxy cohesive
        elements
  A4. Synchronize attributes of ghost nodes
  5 - Update nodal accelerations and velocities
  6 - Update external forces
  7 - Apply boundary conditions to nodes
  8 - Update nodal displacements
```

At first, the results of replicated computations are treated as if they were identical for every mesh partition, but no special care is taken to ensure a stable order in floating-point operations. In *elements-to-node* computations, this can yield slightly different results for different partitions. Thus, to avoid the accumulation of numerical differences along a large number of time steps, numerical attributes of affected topological entities are synchronized sporadically.

In nodal stress computation (item 2.1), temporary results are first computed at each volumetric element from the attributes of the adjacent nodes, which corresponds to the *nodes-to-element* computation pattern. Therefore, the temporary element results are naturally symmetrical, being consistent among different mesh partitions. Next, the element contribution is added to the adjacent nodes, as in the *elements-to-node* computation pattern. In this case, the application calculates nodal results in the traditional way, by traversing the elements of the current mesh partition and then adding the contribution of each element to its adjacent nodes.

Since no global ordering is imposed to element traversals, slightly different numerical results can be obtained for different mesh partitions, due to different local orders of floating-point operations. This requires that proxy entities are sporadically synchronized in order to maintain data consistency among partitions (item A1). If attribute synchronization were completely removed, numerical divergences would propagate along a large number of time steps. On the other hand, numerical differences that do not significantly affect simulation results could be tolerated. Therefore, in order to remove synchronization costs from the simulation, but mitigate the accumulation of numerical divergences, synchronization of proxy entities is done at fixed intervals of n time steps. In our tests, numerical simulations were performed with synchronization intervals of up to 100 time steps, with no significant differences in the achieved results compared to the original results. We observe that, in the numerical simulations considered here, nodal stresses are only required for the determination of fractured facets. Therefore, propagation of numerical inaccuracies is limited to this scope, though divergences on the internal force vector, for example, can propagate longer.

Determination of fractured facets (item 2.2) can be classified by the *nodes-to-element* computation pattern. Thus, assuming that the nodes of each facet are consistent, the computation is naturally symmetric. As a result, synchronization of local sets of fractured facets could be removed if fracture determination in each mesh partition also includes proxy entities. However, numerical divergences in stress computations, even small, can affect the binary decision of whether the fracture criterion has been met, causing the sets of fractured facets to become inconsistent among different partitions. Hence, in order to ensure data consistency, fracture determination remains based on local facets only, followed by a synchronization of the local facet sets (item A2).

Like nodal stresses, the internal force vector (item 3) is computed in two stages, which correspond to the *nodes-to-element* and *elements-to-node* computation patterns respectively. It is directly followed by the cohesive force vector (item 4), which can be decomposed into *nodes-to-element*, *at-element*, and *elements-to-node* computations. *Nodes-to-element* and *at-element* are symmetrical and depend on nodal attributes (e.g. materials and displacement), which remain unchanged to this point in the current time step. *At-element* computations modify and depend on attributes of cohesive elements that were computed in the previous time step (e.g. history information). Since this type of computation is symmetric, the results do not need to be synchronized in order to ensure consistency toward the next time step. Finally, *elements-to-node* computations are based on the results of the other two previous computations, and thus depend on consistent data. The results of *elements-to-node* computations of the internal and cohesive force vectors are treated in a similar manner as nodal stresses. Thus, attributes of proxy nodes are also sporadically synchronized at fixed intervals of n time steps. Only ghost nodes are synchronized every time step, so they can be used by the following computations (items A3 and A4).

The remaining computations (items 5 to 8) follow the *at-node* pattern. Hence, they are naturally symmetric and do not require additional synchronization.

4.7.3. Parallel approach based on symmetrical computations

The sporadic synchronization of proxy nodes, employed by the previous parallel approach, can be removed from the numerical application. This is accomplished by replacing regular iterators by the corresponding stable iterators for the traversal of elements in *elements-to-node* computations (Section 4.4). As a result, nodal stresses, and internal and cohesive forces are computed in a symmetrical fashion, thus yielding numerically consistent results for local and proxy nodes for different mesh partitions, with no inter-partition synchronization. With fracture determination performed for both local and proxy facets, the synchronization of local sets of fractured facets can also be removed. On the other hand, proxy facets may be adjacent to ghost nodes and thus those nodes must also be synchronized before the fracture determination. The structure of the parallel code based on this approach is shown in Algorithm 4. It only requires the synchronization of ghost nodes (items A1 and A2). One advantage of reducing the number of synchronization points using symmetrical computations is the structure of the parallel simulation is simplified.

Algorithm 4. Basic structure of the parallel fracture simulation using "symmetrical computations" based on stable element iterators.

```
Initialize the finite element model

For each time step
  1 - Compute current applied boundary conditions
  2 - Check the insertion of new cohesive elements
    2.1 - Compute nodal stresses
    A1. Synchronize attributes of ghost nodes
    2.2 - Check fracture criteria along facets
    2.3 - Insert cohesive elements
    2.4 - Update nodal masses
  3 - Compute the internal force vector
  4 - Compute the cohesive force vector
  A2. Synchronize attributes of ghost nodes
  5 - Update nodal accelerations and velocities
  6 - Update external forces
  7 - Apply boundary conditions to nodes
  8 - Update nodal displacements
```

4.7.4. Mixed parallel approach

By employing symmetrical computations, the number of attribute synchronization points is reduced with respect to the conventional approach based on local computations. Sometimes, though, a combined strategy may become more advantageous. Regarding the present numerical fracture simulation, synchronization of local sets of fractured facets is not necessary when computations are symmetrical. This requires that attributes of ghost nodes be synchronized before the fracture criterion can be evaluated. However, a substantially smaller amount of data is expected to be exchanged by neighboring partitions during synchronization of facet sets compared with attributes of ghost nodes. Hence, in order to reduce data transferring among partitions, we propose a mixed approach. We employ symmetrical operations but maintain fracture determination for local facets only, with a subsequent synchronization of the corresponding fractured facet sets. The updated structure of this parallel simulation is presented in Algorithm 5.

Algorithm 5. Basic structure of the parallel fracture simulation based on the mixed approach, with stable iterators and synchronization of local sets of fractured facets.

```
Initialize the finite element model

For each time step
  1 - Compute current applied boundary conditions
  2 - Check the insertion of new cohesive elements
    2.1 - Compute nodal stresses
    2.2 - Check fracture criteria along facets
    A1. Synchronize local sets of fractured facets
    2.3 - Insert cohesive elements
    2.4 - Update nodal masses
  3 - Compute the internal force vector
  4 - Compute the cohesive force vector
  5 - Update nodal accelerations and velocities
  6 - Update external forces
  7 - Apply boundary conditions to nodes
  8 - Update nodal displacements
  A2. Synchronize attributes of ghost nodes
```

5. Computational experiments

In order to assess the correctness, efficiency, and scalability of parallel simulations based on the proposed topological framework, we have performed a set of computational experiments. The experiments were executed on Intel 64 Cluster Abe, located at the National Center for Supercomputing Applications (NCSA). Each node of the cluster is composed of two 2.33 GHz quad core Intel 64 processors (8 cores per node), with 1 GB RAM per core. Nodes are interconnected through an InfiniBand network, with a Red Hat Enterprise Linux 4 (2.6.18), using gcc v.3.4.6 compiler. Exactly one mesh partition is assigned to each processor core.

5.1. Scalability of cohesive element insertion

The following experiment demonstrates the scalability of the parallel topological framework regarding the insertion of cohesive elements. For this purpose, the computations were decoupled from the mechanics analysis. This complements Ref. [10], which suggests that the ParTopS framework should scale well, but does not provide results for large mesh sizes and number of processors. The experiment measures the ability of the parallel framework to tackle larger problems efficiently, by varying the number of processors with mesh size in order to maintain the same level of efficiency. For the ideal case, execution time is expected to remain constant if the number of processors increases proportionally with the number of elements. In this experiment, we have used the sample three-dimensional grid of Fig. 6. Mesh partition size corresponds to approximately $50 \times 50 \times 50$ hexahedral cells decomposed into six linear tetrahedral elements (Tet4) each, or 750,000 elements per partition. Cohesive elements are incrementally inserted at approximately 50% of the internal facets, along 50 execution steps. At each step, 1% of the facets are randomly chosen and cohesive elements inserted at them.

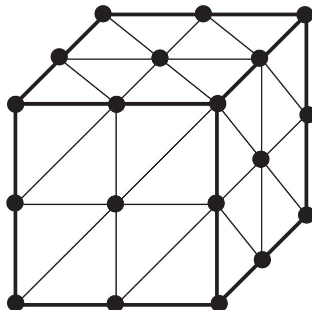

Fig. 6. Sample tetrahedral grid used in the computational experiment to evaluate scalability of cohesive element insertion. Different mesh discretizations are employed.

Table 1
Total time for parallel insertion of cohesive elements.

Grid size	# Elements (millions)	# Processors	Total time (s)
$50 \times 50 \times 50$	0.75	1 (serial)	16.59
$100 \times 100 \times 100$	6.00	8	24.30
$200 \times 200 \times 200$	48.00	64	28.12
$400 \times 400 \times 400$	384.00	512	29.24
$450 \times 450 \times 450$	546.75	729	31.41
$500 \times 500 \times 500$	750.00	1000	31.74

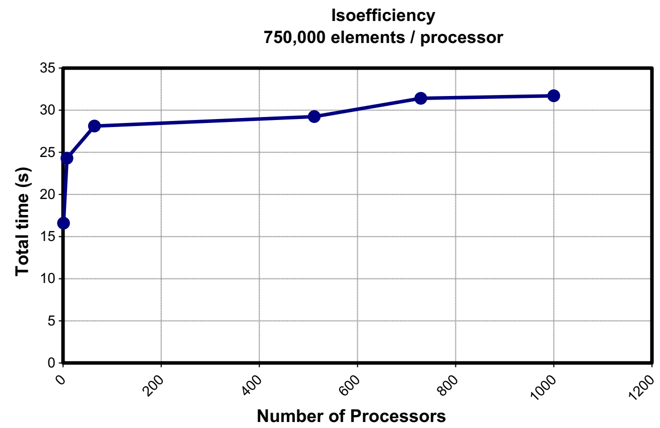

Fig. 7. Total execution time for 50 steps of parallel insertion of cohesive elements into the tetrahedral grid versus the number of processor cores utilized. Mesh discretization increases proportionally with the number of processors.

Results for the various mesh sizes and numbers of processors are summarized in Table 1, and the total execution times versus the number of processors are plotted in Fig. 7. For a large number of processors, the total times show a constant trend, with reduced variations compared with the use of a small number of processors. Consequently, the parallel algorithm to insert cohesive elements presents the expected scalability results. The comparison with the serial simulation shows the small parallel overhead introduced to the original application. For 1000 processors, the execution time is approximately 1.9 times the serial time; however, the problem size that can be solved is approximately 1000 times larger.

5.2. Parallel fracture simulations

Efficiency, scalability and generality of fracture simulation were evaluated by a set of computational tests on finite element models for various mesh sizes and number of partitions. For simplicity, a Mode I fracture test is employed with a predefined crack path, and the geometry and boundary conditions of the models are illustrated in Fig. 8. Additionally, a microbranching fracture test is also performed to demonstrate the generality of the proposed framework. Within a rectangular PMMA plate, an initial strain (ε_0) of 0.036 and 0.043 is applied for a predefined crack path example and a microbranching fracture test, respectively. Such geometry and boundary conditions are motivated by the original experiment work by Sharon and Fineberg [40]. Both 2D and 3D models are

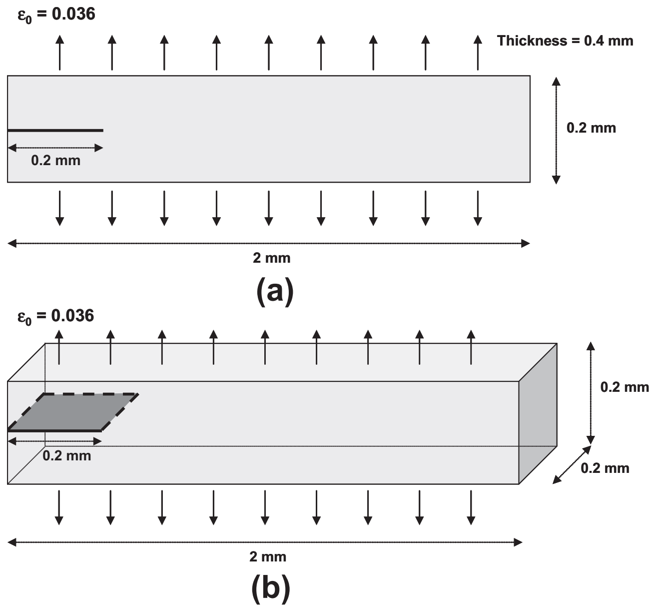

Fig. 8. Geometries of the 2D (a) and 3D (b) models used in the computational simulation experiments.

tested. The 2D model is discretized into quadratic triangular elements (T6) while the 3D model is discretized into linear tetrahedron elements (Tet4). Mesh partitions are created by using the METIS mesh partitioner [24,25]. In addition, for the 2D predefined crack path model, the total simulated time is 2 μs, in 10,000 steps of 0.2 ns. For the 3D predefined crack path model, the total simulated time is 2.4 μs, in 12,000 steps of 0.2 ns. For the microbranching example, the total simulated time is 22 μs, in 220,000 steps of 0.1 ns. Note that the selection of the mesh sizes and time step sizes is based on the work by Zhang et al. [1] and Park et al. [43].

Material properties are obtained from Ref. [1,30,43]. For elastic material properties of PMMA, elastic modulus is 3.24 GPa, Poisson's ratio is 0.35, and density is 1190 kg/m^3. For the Mode I fracture parameters of the predefined crack path examples, the fracture energy (ϕ_n) and the cohesive strength ($\sigma_{\max}$) are 352 N/m and 324 MPa, respectively, as in Ref [43]. For the microbranching example, the fracture energy (ϕ_n) and the cohesive strength ($\sigma_{\max}$) are 352.3 N/m and 129.6 MPa, respectively, which corresponds to the properties of PMMA in Ref [1,30]. The Mode II fracture parameters are assumed to be the same as the Mode I parameters. Additionally, the PPR model [20] is utilized to define the cohesive traction-separation relationship. In the PPR model, the shape parameter (α) is selected as 2, which leads to the linear softening.

Initially null velocities and accelerations ($\vec{v} = 0$, $\vec{a} = 0$), and an initial strain (ε_0) are imposed to nodes along the upper and lower model boundaries. The corresponding initial nodal displacements vary proportionally along the vertical distance from the center of the model. For cohesive fracture simulation, cohesive elements are inserted when the averaged normal stress along a facet is greater than the normal cohesive strength (i.e. a stress-based criterion). Note that the criterion is checked at every 10 simulation steps.

Table 3
Execution times, in seconds, for 10,000 simulation steps considering the 2D model. The mesh is discretized into 400 × 40 regular quadrilaterals, each divided into four T6 triangular elements (i.e. total of 64,000 elements). Total time is the sum of all the individual time steps. The execution time of a simulation step corresponds to the time required by numerical computations plus the time for synchronization of attributes and fractured facets, and insertion of cohesive elements.

# Processors	Time (seconds)		
	Synchronization	Cohesive insertion	Total
1 (serial)	0.00	0.02	**6333.35**
2	18.70	0.80	**3572.79**
4	21.12	4.22	**1948.94**
8	25.87	6.87	**1402.29**
16	32.33	11.08	**659.06**

Table 4
Performance metrics for the numerical simulations of the 2D model. The mesh is discretized into 400 × 40 regular quadrilaterals, each divided into four T6 triangular elements (i.e. total of 64,000 elements). *Speedup* is defined as the ratio between sequential and parallel execution times, and *efficiency* (or *processor utilization*) is the speedup divided by the number of processors [26]. The percentage of the total time spent in attribute synchronization is also presented.

# Processors	Metric		
	Speedup	Efficiency	Synchronization (% total time)
1 (serial)	–	–	0.00
2	1.77	0.89	0.52
4	3.25	0.81	1.08
8	4.52	0.56	1.84
16	9.61	0.60	4.90

5.2.1. Comparison of parallel approaches

This computational experiment compares the four parallel approaches for attribute synchronization discussed in Section 4.7. To this end, the 3D model was discretized into a 400 × 40 × 40 grid of regular hexahedral cells, and each cell is divided into 6 linear tetrahedral elements (i.e. total of 3,840,000 elements). The tetrahedral mesh was decomposed into 32 partitions, and 12,000 time steps were simulated.

Table 2 shows the computational times corresponding to the synchronization of numerical attributes and fractured facets requested by the application, for each parallel approach. Synchronization times for the approaches with replicated computations were substantially lower than the conventional approach based on local computations, as a consequence of the reduced number of synchronization points of the parallel application. The best achieved performance corresponds to the mixed approach, which

Table 2
Execution times, in seconds, for 12,000 simulation steps of the 3D model discretized into 3,840,000 linear tetrahedra (Tet4), on 32 processor cores (1 partition per core), using different approaches for attribute synchronization.

Parallel approach	Time (seconds) Synchronization (attributes + facets)
Local computations (conventional approach)	1516.39
Replicated computations (with sporadic synchronization of proxy nodes)	1113.72
Replicated symmetrical computations (with stable iterators)	1120.04
Mixed approach (symmetrical and local computations)	1104.01

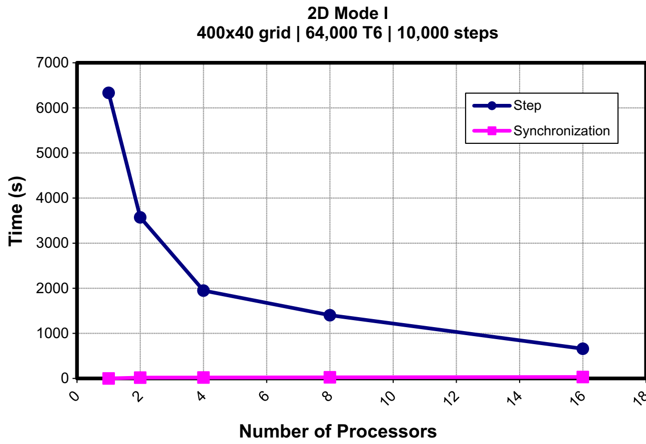

Fig. 9. Total execution times of the simulation of the 2D model versus the number of processors utilized. The analysis attribute synchronization times are also plotted.

requires only two synchronization points. In this case, synchronization times were reduced by approximately 27% compared with the local computation approach. The approach based on symmetrical computations with stable iterators (Section 4.7.3) resulted in synchronization times slightly greater than the mixed approach, due to the higher cost of synchronization of ghost nodes compared with the fractured facet sets. The approach based on replicated computations with sporadic synchronization of proxy nodes (Section 4.7.2) showed synchronization times very close to the mixed approach. Both approaches have an equivalent number of synchronization points, when the residual cost of sporadic proxy synchronizations is negligible.

5.2.2. Parallel performance compared with serial simulations

This experiment compares the performance of parallel simulations with the serial version, for an increasing number of processors. Hence, mesh size is fixed, whereas the number of processors varies. In order to have the simulation also executed on a single processor, the 2D model was employed. The mesh is discretized into 400×40 regular quadrilaterals, each divided into four T6 triangular elements (i.e. total of 64,000 elements). The number of simulation steps is equal to 10,000. The mixed parallel approach was used in this experiment and in the others that follow.

The achieved results are presented in Table 3, and some performance metrics are shown in Table 4. In Fig. 9, we plot both the total execution time and the time spent on attribute synchronization with respect to the number of processors. For the mesh discretization utilized, the most significant performance gains of the parallel simulation with regard to the serial version occur on up to four processors in this example (e.g. more than 80% of efficiency). An increase in the number of processors does not yield the same proportional benefits. This happens because communication costs, which are dominated by attribute synchronization, tend to increase, while less computation is done per processor within each simulation step. The mesh partitioning for 16 processors and corresponding numerical results for σ_y are shown in Fig. 10.

5.2.3. Microbranching fracture simulations

Results of two-dimensional parallel microbranching fracture simulation [1,20] are shown in Fig. 11. Fracture propagation is based on mixed-mode of fracture and the extrinsic cohesive zone model [22,23,20]. The constitutive model used is PPR [20]. This type of simulation allows for complex fracture patterns as the ones observed in the experiment by Sharon and Fineberg [40], which demonstrates that the proposed parallel approach is general enough for both fracture and microbranching simulations.

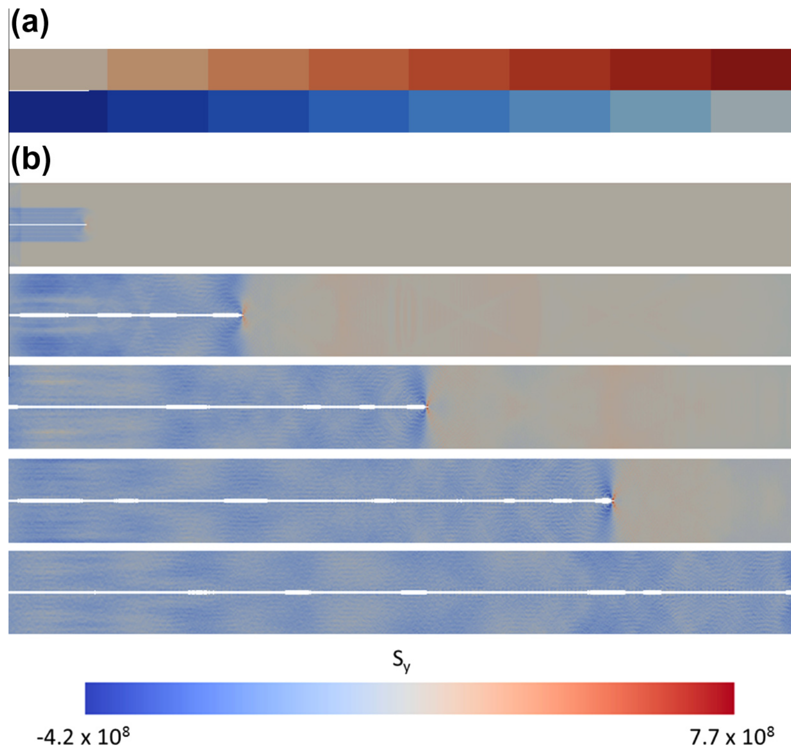

Fig. 10. (a) Mesh partitioning of the 2D model for 16 processors; (b) numerical results, obtained at steps 100, 2500, 5000, 7500 and 10,000 ($S_y \equiv \sigma_y$).

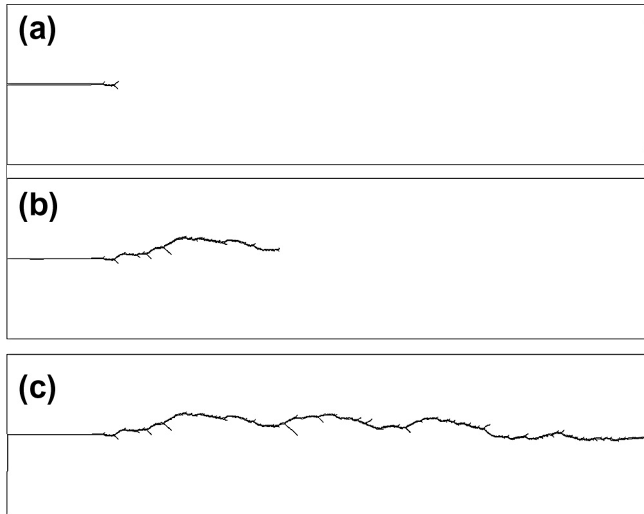

Fig. 11. Fracture evolution over time, at time steps: (a) 20,000 (2 μs); (b) 80,000 (8 μs); (c) 220,000 (22 μs).

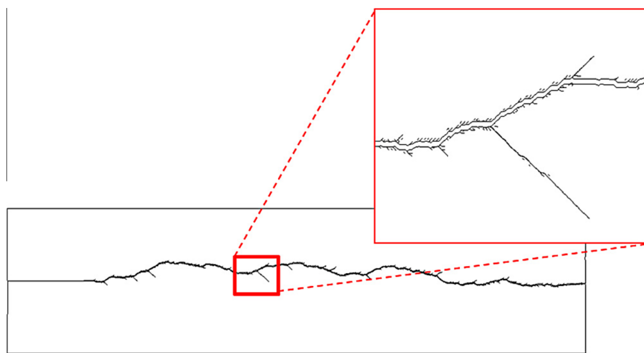

Fig. 12. A region around the main crack path is enlarged to show microbranching phenomena.

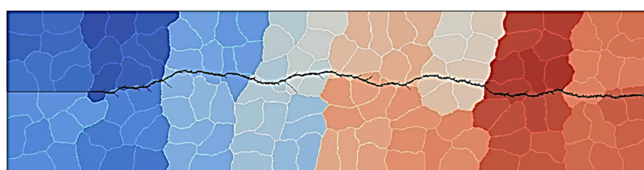

Fig. 13. The finite element model decomposed into 128 partitions.

The geometry used in the parallel simulation is similar to Fig. 8(a), a rectangular plate with an initial notch. The geometric domain, though, has dimensions equal to 128 mm × 32 mm, and the length of the initial notch is 32 mm. A unit thickness is assigned for the computation of plane stresses. The rectangular geometry corresponds to the one used by Zhang et al. [1] to investigate microbranching fracture phenomena in brittle materials by using the extrinsic cohesive model.

The finite element model was discretized into a bidimensional mesh of quadratic triangular elements (T6), and consists of 2,359,296 elements and 4,722,817 nodes. Initial positions of internal nodes were randomly perturbed by a factor of 0.3 times the minimum distance of a vertex of each element to the adjacent vertices, as suggested by Ref. [42], in order to reduce mesh bias and improve crack path representations. Note that the nodal perturbation helps reducing the error between the total length of the achieved fracture path with respect to the expected length. A

Table 5
Execution times, in seconds, for 12,000 simulation steps of the 3D model. The mesh is discretized into 400 × 40 × 40 hexahedra, each hexahedron is divided into 6 linear tetrahedral elements (i.e. total of 3,840,000 elements). The total time is the sum of all the time steps. The time of each step corresponds to the numerical computation time plus the time for synchronization of attributes and fractured facets, and insertion of cohesive elements.

# Processors	Time (seconds)		
	Synchronization	Cohesiveinsertion	Total
32	1104.01	271.22	24470.57
64	955.50	214.14	13311.10
128	914.96	298.81	7481.55
256	536.10	136.97	4124.10
512	358.67	105.63	2365.07

Table 6
Performance metrics for the numerical simulations of the 3D model. The mesh is discretized into 400 × 40 × 40 hexahedra, each hexahedron is divided into 6 linear tetrahedral elements (i.e. total of 3,840,000 elements). *Relative speedup* is the ratio between current parallel execution time and initial simulation on 32 processors. *Efficiency* (or *processor utilization*) is defined as the speedup divided by the number of processors [26]. The percentage of the total time spent in attribute synchronization is also presented.

# Processors	Metric Relative speedup	Efficiency	Synchronization (% total time)
32	–	–	4.51
64	1.84	0.92	7.18
128	3.27	0.82	12.23
256	5.93	0.74	13.00
512	10.35	0.65	15.17

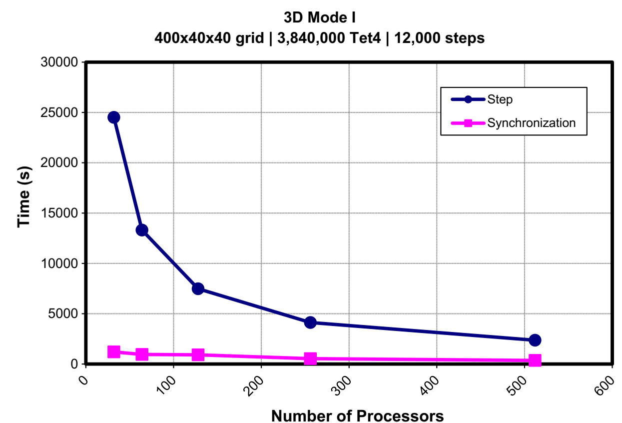

Fig. 14. Total execution times of the simulation of the 3D model versus the number of processors utilized. The analysis attribute synchronization times are also shown.

Laplacian smoothing operator was also employed in order to improve quality of elements in the mesh [42].

This numerical experiment was performed on a small cluster, with 13 machines connected by a Gigabit Ethernet network. Each machine has one Intel(R) Pentium(R) D processor, with two cores at 3.40 GHz, 2 GB of RAM, 64-bit Red Hat Linux 4.3.2-7 operating system (kernel version 2.6.27) and gcc compiler v. 4.3.2. The finite element mesh was decomposed into 128 partitions, using the graph partitioner METIS [24,25]. The mesh partitions were assigned to the physical processors available at the time of the simulation (13 machines, or 26 processor cores, with approximately 5 partitions per core).

The fracture propagation pattern shown in Fig. 11 is consistent with the results obtained in Ref. [1] for serial simulations using models of reduced geometry. The main crack branch develops

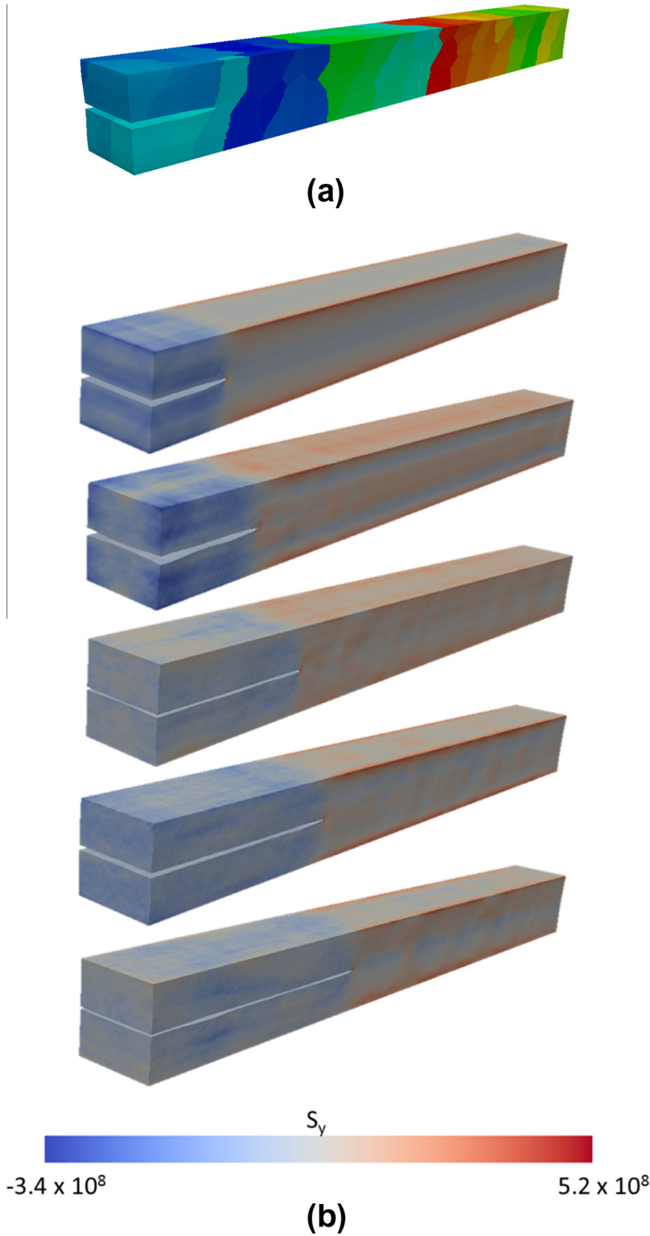

Fig. 15. (a) Mesh partitioning of the 3D model, for 128 processors; (b) numerical simulation results for steps 500, 3000, 6000, 9000 and 12,000 ($S_y \equiv \sigma_y$).

along the horizontal direction near the center of the model. Along the fracture path, a number of micro-cracks occur spontaneously, starting from the main branch, as dictated by the problem physics.

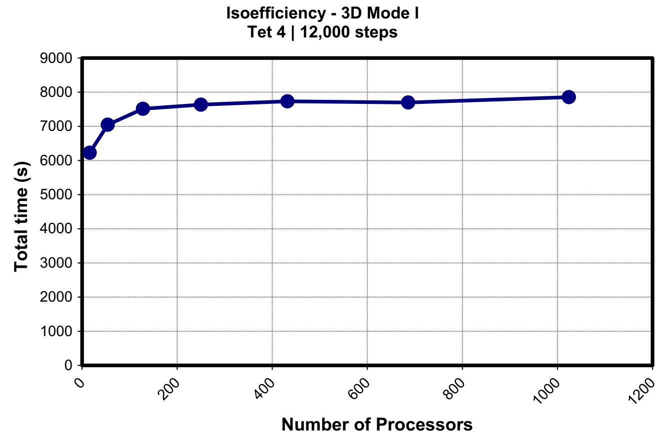

Fig. 16. Total execution time for the 3D model versus the number of processors utilized. Mesh size is proportional to the number of processors.

In Fig. 12, the region around the main crack branch is enlarged to show the secondary micro-cracks. Branching frequency and size tend to increase with the initial deformation applied to the model, as observed in Ref. [1]. Mesh partitioning is shown in Fig. 13.

5.2.4. Relative parallel performance

The performance of parallel simulations is evaluated with respect to an increasing number of processors, considering a large number of processors. As in the previous experiment, mesh discretization is fixed, whereas the number of processors varies. The 3D model was used in this experiment; the mesh is discretized into $400 \times 40 \times 40$ hexahedra, each hexahedron is divided into 6 linear tetrahedral elements (i.e. total of 3,840,000 elements). The number of processors varies in the range between 32 and 512, and 12,000 simulation steps were employed.

Execution times are summarized in Table 5, and performance metrics in Table 6. Fig. 14 plots both the total execution time and the time spent on attribute synchronization versus the number of processors. For this problem size, the most significant performance gains are achieved on up to 128 processors (e.g. more than 80% of efficiency), compared with the initial execution on 32 processors, or four times the initial number of processors. When more processors are utilized, communication, which is mainly represented by attribute synchronization, requires a greater percentage of the total time step, as expected, and thus proportional performance benefits are no longer obtained. The mesh partitioning for 128 processors and the corresponding numerical results for σ_y are illustrated in Fig. 15.

Table 7
Total execution times, in seconds, for the various mesh sizes and number of processors tested. A total of 12,000 simulation steps were performed for the 3D model. The total time is the sum of all the time steps, and corresponds to the numerical computation time plus the time for synchronization of attributes and fractured facets, and insertion of cohesive elements.

Model			Time (s)		
Mesh size	Number of elements	Number of processors	Synchronization (attrib + facets)	Cohesiveinsertion	Total time
$200 \times 20 \times 20$	480,000	16	377.18	40.11	6226.62
$300 \times 30 \times 30$	1,620,000	54	731.10	158.05	7046.93
$400 \times 40 \times 40$	3,840,000	128	927.37	293.61	7513.59
$500 \times 50 \times 50$	7,500,000	250	957.49	325.30	7634.65
$600 \times 60 \times 60$	12,960,000	432	844.09	228.37	7732.84
$700 \times 70 \times 70$	20,580,000	686	886.71	285.48	7697.88
$800 \times 80 \times 80$	30,720,000	1024	908.54	321.82	7854.31

5.2.5. Scalability

This experiment measures the ability of the parallel simulation to solve larger problems, considering that the number of processors increases at the same rate as the problem size (*weak scaling*). Therefore, the number of elements per processor is maintained nearly constant while the number of processors varies, which leads to a proportional increase in the problem size. In this numerical experiment, we use the 3D model of Fig. 8(b) with 12,000 simulation steps. The number of elements per processor is fixed as 30,000 while the number of processors and the number of elements increase.

Results for various numbers of processors are presented in Table 7, and the total simulation time versus the number of processors is plotted in Fig. 16. In the ideal case, the number of processors required to solve a problem is expected to scale linearly with problem size, in order to maintain initial efficiency. Therefore, the total execution time should be constant whenever model size and number of processors increase at the same rate, especially for a large number of processors. If this is the case, the application can be considered to scale linearly with respect to problem size. This corresponds to the isoefficiency scalability metric [26,41].

In Fig. 16, we observe a significant positive change on the simulation time for the smaller models (and the corresponding number of processors). However, as the number of processors (and problem size) increases, the total simulation time is maintained nearly constant. Therefore, the parallel simulation is considered scalable for the problem sizes and number of processors tested. Achieved results are consistent with the scalability for the insertion of cohesive elements obtained in Section 5.1.

6. Concluding remarks

The ParTopS topological framework has been extended in order to achieve scalable parallel dynamic computational simulations of large scale cohesive fracture problems. Topological support is provided for extrinsic cohesive zone models, through efficient adaptive insertion of cohesive elements.

In order to evaluate the applicability of the framework to realistic fracture problems, a dynamic fracture simulation code has been parallelized. Starting from a serial code developed based on the TopS data structure, we discuss four different approaches to parallelize the code with minimal impact on application re-coding. In this study, the proposed approach, which combines local computations and replicated computations with stable iterators, is the most efficient.

Computational experiments demonstrate the scalability of the parallel dynamic fracture code for both 2D and 3D simulations. The results achieved show the applicability of the proposed framework in simulations of relatively large models. Simulations have scaled to a large number of processors and good overall computational efficiency was achieved.

Some important issues, like load balancing, have not been addressed here, though. The load imbalance introduced by insertion of new cohesive elements is assumed to be small compared to the computational load for processing the bulk elements. We also assume that the initial mesh is well balanced, which is ensured by the initial mesh partitioning method. Support for adaptive mesh refinement (e.g. *h-refinement*) and coarsening is another important feature to consider in order to improve the efficiency of numerical simulations, as discussed in Ref [43]. This will be the focus of a follow-up work. However, with the current state of proposed topological framework, it can be used to solve real scale fracture problems, which is important to avoid artifacts due to reduced fracture models (as discussed in the introduction).

Acknowledgements

We acknowledge support from the US National Science Foundation (NSF) through Grant CMMI #1321661. This work used TeraGrid Resources under Grant TG-ASC050039N. RE and WC thank CNPq (Brazilian National Research and Development Council) for the financial support to conduct this research. GHP is thankful to the Donald B. and Elisabeth M. Willett endowment at the University of Illinois at Urbana-Champaign (UIUC). KP acknowledges support from the National Research Foundation (NRF) of Korea through Grant #2011-0013393. The authors would also like to extend their appreciation to Ms. Sofie Leon for her invaluable input to this publication. 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] Z. Zhang, G.H. Paulino, W. Celes, Extrinsic cohesive modeling of dynamic fracture and microbranching instability in brittle materials, Int. J. Numer. Methods Eng. 72 (8) (2007) 893–923.
[2] K.D. Papoulia, S.A. Vavasis, P. Ganguly, Spatial convergence of crack nucleation using a cohesive finite-element model on a pinwheel-based mesh, Int. J. Numer. Methods Eng. 67 (1) (2006) 1–16.
[3] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, L. Kale, ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications, Eng. Comput. 22 (3–4) (2006) 215–235.
[4] J.-F. Remacle, O. Klaas, J.E. Flaherty, M.S. Shephard, Parallel algorithm oriented mesh database, Eng. Comput. 18 (3) (2002) 274–284.
[5] E.S. Seol, M.S. Shephard, Efficient distributed mesh data structure for parallel automated adaptive analysis, Eng. Comput. 22 (3–4) (2006) 197–213.
[6] C. Ozturan, Distributed Environment and Load Balancing for Adaptive Unstructured Meshes, PhD Thesis, Computer Science Department, Rensselaer Polytechnic Institute, 1995.
[7] C. Ozturan, H.L. de Cougny, M.S. Shephard, J.E. Flaherty, Parallel adaptive mesh refinement and redistribution on distributed memory computers, Comput. Methods Appl. Mech. Eng. 119 (1–2) (1994) 123–127.
[8] B.S. Kirk, J.W. Peterson, R.H. Stogner, G.F. Carey, LibMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations, Eng. Comput. 22 (3) (2006) 237–254.
[9] J.R. Stewart, H.C. Edwards, A framework approach for developing parallel adaptive multiphysics applications, Finite Elem. Anal. Des. 40 (12) (2004) 1599–1617.
[10] R. Espinha, Celes. Waldemar, N. Rodriguez, G.H. Paulino, ParTopS: compact topological framework for parallel fragmentation simulations, Eng. Comput. 25 (4) (2009) 345–365.
[11] W. Celes, G.H. Paulino, R. Espinha, A compact adjacency-based topological data structure for finite element mesh representation, Int. J. Numer. Methods Eng. 64 (11) (2005) 1529–1565.
[12] W. Celes, G.H. Paulino, R. Espinha, Efficient handling of implicit entities in reduced mesh representations, J. Comput. Inf. Sci. Eng. 5 (4) (2005) 348–359 (Special Issue on Mesh-Based Geometric Data Process).
[13] G.H. Paulino, W. Celes, R. Espinha, Z. Zhang, A general topology-based framework for adaptive insertion of cohesive elements in finite element meshes, Eng. Comput. 24 (1) (2008) 59–78.
[14] I. Foster, Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, Addison-Wesley, Boston, 1995.
[15] MPI Forum, MPI: A Message-Passing Interface Standard, 2010. <http://www.mpi-forum.org>.
[16] L.V. Kalé, S. Krishnan, CHARM++: a portable concurrent object oriented system based on C++, in: A. Paepcke (Ed.), Proceedings of OOPSLA'93 September 1993, ACM Press, 1993, pp. 91–108.
[17] L.V. Kalé, S. Krishnan, Charm++: parallel programming with message-driven objects, in: G.V. Wilson, P. Lu (Eds.), Parallel Programming Using C++, MIT Press, London, 1996, pp. 175–213.
[18] G. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000.
[19] K. Park, Potential-Based Fracture Mechanics Using Cohesive Zone and Virtual Internal Bond Modeling, PhD Dissertation, Department of Civil and Environmental Engineering, University of Illinois at Urbana-Champaign, 2009.
[20] K. Park, G.H. Paulino, J.R. Roesler, A unified potential-based cohesive model of mixed-mode fracture, J. Mech. Phys. Solids 57 (6) (2009) 891–908.
[21] T. Belytschko, W.K. Liu, B. Moran, Nonlinear Finite Elements for Continua and Structures, Wiley, New York, 2000.
[22] G.T. Camacho, M. Ortiz, Computational modelling of impact damage in brittle materials, Int. J. Solids Struct. 33 (20–22) (1996) 2899–2938.

[23] M. Ortiz, A. Pandolfi, Finite-deformation irreversible cohesive elements for three-dimensional crack-propagation analysis, Int. J. Numer. Methods Eng. 44 (9) (1999) 1267–1282.

[24] G. Karypis, V. Kumar, METIS – Serial Graph Partitioning and Fill-reducing Matrix Ordering Library, Department Computer Science Engineering, University of Minnesota, 1995. <http://www.cs.umn.edu/~karypis/metis>.

[25] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, J. Parallel Distrib. Comput. 48 (1) (1998) 96–129.

[26] M.J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, New York, 2004.

[27] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, C. Vaughan, Zoltan data management services for parallel dynamic applications, Comput. Sci. Eng. 4 (2) (2002) 90–97.

[28] S. Balay, W.D. Gropp, L.C. McInnes, B.F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A.M. Bruaset, H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhauser Press, 1997, pp. 163–202.

[29] B. Carter, C.S. Chen, L.P. Chew, N. Chrisochoides, G.R. Gao, G. Heber, A.R. Ingraffea, R. Krause, C. Myers, D. Nave, K. Pingali, P. Stodghill, S. Vavasis, P.A. Wawrzynek, Parallel FEM simulation of crack propagation – challenges, status, and perspectives, Lecture Notes in Computer Science, vol. 1800, Springer, Berlin, Heidelberg, 2000, pp. 443–449 (Parallel and Distributed Processing).

[30] X.P. Xu, A. Needleman, Numerical simulations of fast crack growth in brittle solids, J. Mech. Phys. Solids 42 (9) (1994) 1397–1434.

[31] A. Pandolfi, M. Ortiz, Solid modeling aspects of three-dimensional fragmentation, Eng. Comput. 14 (4) (1998) 287–308.

[32] A. Pandolfi, M. Ortiz, An efficient adaptive procedure for three-dimensional fragmentation simulations, Eng. Comput. 18 (2) (2002) 148–159.

[33] A. Mota, J. Knap, M. Ortiz, Fracture and fragmentation of simplicial finite element meshes using graphs, Int. J. Numer. Methods Eng. 73 (11) (2008) 1547–1570.

[34] I. Dooley, S. Mangala, L. Kale, P. Geubelle, Parallel simulations of dynamic fracture using extrinsic cohesive elements, J. Sci. Comput. 39 (1) (2009) 144–165.

[35] S. Mangala, T. Wilmarth, S. Chakravorty, N. Choudhury, L.V. Kalé, P.H. Geubelle, Parallel adaptive simulations of dynamic fracture events, Eng. Comput. 24 (4) (2008) 341–358.

[36] R. Radovitzky, A. Seagraves, M. Tupek, L. Noels, A scalable 3D fracture and fragmentation algorithm based on a hybrid, discontinuous Glerkin, cohesive element method, Comput. Methods Appl. Mech. Eng. 200 (1–4) (2011) 326–344.

[37] L. Noels, R. Radovitzky, A general discontinuous Galerkin method for finite hyperelasticity. formulation and numerical applications, Int. J. Numer. Methods Eng. 68 (1) (2006) 64–97.

[38] L. Noels, R. Radovitzky, An explicit discontinuous Galerkin method for non-linear solid dynamics: formulation, parallel implementation and scalability properties, Int. J. Numer. Methods Eng. 74 (9) (2008) 1393–1420.

[39] M. Heath, Scientific Computing: An Introductory Survey, second ed., McGraw-Hill, 2002.

[40] E. Sharon, J. Fineberg, Microbranching instability and the dynamic fracture of brittle materials, Phys. Rev. B 54 (10) (1996) 7128–7139.

[41] A.Y. Grama, A. Gupta, V. Kumar, Isoefficiency: measuring the scalability of parallel algorithms and architectures, IEEE Parallel Distrib. Technol. 1 (3) (1993) 12–21.

[42] G.H. Paulino, K. Park, W. Celes, R. Espinha, Adaptive dynamic cohesive fracture simulation using nodal perturbation and edge-swap operators, Int. J. Numer. Methods Eng. 84 (2010) 1303–1343.

[43] K. Park, G.H. Paulino, W. Celes, R. Espinha, Adaptive mesh refinement and coarsening for cohesive zone modeling of dynamic fracture, Int. J. Numer. Methods Eng. 92 (1) (2012) 1–35.

[44] K. Park, G.H. Paulino, Parallel computing of wave propagation in three dimensional functionally graded media, Mech. Res. Commun. 38 (6) (2011) 431–436.

[45] K.R. Shah, B.J. Carter, A.R. Ingraffea, Hydraulic fracturing simulation in parallel computing environment, Int. J. Rock Mech. Min. Sci. 34 (1997) 3–4.

[46] H. Bao, J. Bielak, O. Ghattas, L.F. Kallivokas, D.R. O'Hallaron, J.R. Shewchuk, J. Xu, Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers, Comput. Methods Appl. Mech. Eng. 152 (1998) 85–102.

[47] A. Ural, G. Heber, P.A. Wawrzynek, A.R. Ingraffea, D.G. Lewicki, J.B. Cavalcante-Neto, Three-dimensional, parallel, finite element simulation of fatigue crack growth in a spiral bevel pinion gear, Eng. Fract. Mech. 72 (2005) 1148–1170.

[48] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L.C. Wilcox, S. Zhong, Scalable adaptive mantle convection simulation on petascale supercomputers, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, 2008, p. 62.