5
Three-dimensional Distributed Cohesive Fracture and Fragmentation Simulations

In this chapter, we focus on the implementation of distributed three-dimensional cohesive fracture and fragmentation simulation with no adaptivity (no mesh refinement and coarsening). This imposes great challenge because now we are dealing with 3D meshes, and thus the simulation is even more complex due to more memory consumption and more elaborate algorithms to deal with insertion of cohesive elements. Therefore, we need a simple and efficient data structure to continue handling a large number of global memory accesses. The data structure is extended to 3D entities, which includes tetrahedron elements and linear triangular cohesive elements. Because the nodal neighborhood increases, the number of colors in the mesh greatly increases, which leads to launching more kernels than the two-dimensional simulation case.

This chapter is organized as follows: first, we describe theoretically how we represent the distributed mesh and the communication layer and then give details on the data structure used to implement it. We then describe how we insert cohesive elements in a 3D mesh on a single GPU, and then extend it to a distributed mesh space. We explain the distrbuted simulation itself and nodal synchronization. Finally, we present details of our numerical results.

5.1
Distributed mesh and communication layer representation

In this work, we deal with massive three-dimensional mesh sizes. An example of such mesh is the 3D beam discretized into x = 114, y = 39, and z = 12 and refined in the middle until a level two refinement (over 2.3 million elements), shown in Figure 5.1, which is run in our experiments. Our goal is to distribute these meshes, which are too large to fit into a single GPU or too costly to be runned by one, into several nodes of a computer cluster, each containing one Graphical Processing Unit. Each node is responsible for simulating part of the distributed mesh on its GPU. Figure 5.1a shows the initial mesh.

During our finite elements analysis, entities such as nodes and elements need to access data stored in adjacent entities. For example, computing the nodal mass requires contribution from adjacent elements. However, adjacent elements can belong to other partitions, so we need to communicate with them to obtain the necessary data. We create a comunnication layer that is composed

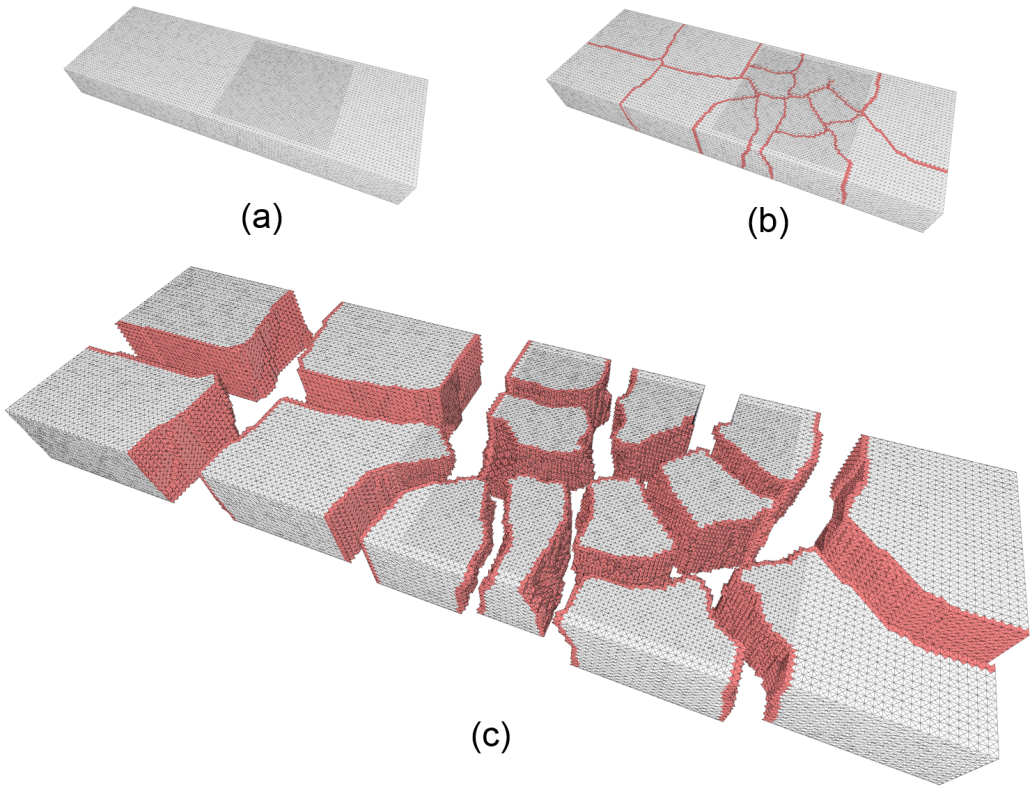

Figure 5.1 – 3D Beam mesh with over 2.3 million bulk elements in (a) its original form; (b) discretized into 15 mesh parts; and (c) showing the communication layer for each partition in red.

by copies of all remote adjacent elements to the nodes of the current partition's boundary, plus these elements' nodes. The communication layer is used for two purposes: (a) perform computations on boundary entities and (b) synchronize topological entities and data from the analysis. Figures 5.1b and 5.1c show the mesh partitioned and the communication layer in red used to synchronize the attributes of the simulation.

We follow Espinha et al's [74] proposal for classification of topological entities used in ParTops. The partitioned mesh is divided into interior part and the communication layer. The interior part contains elements that belong only to that partition (i.e., this is the owner partition). We call these elements *local* elements. The communication layer contains "copy" of elements that belong to other partitions. We call these elements *proxy* elements. Local and proxy elements can be either bulk elements or cohesive elements. Bulk elements remain topologically unchanged during the entire simulation, while cohesive elements are constantly being added to the mesh "on-the-fly" between bulk elements on facets. Nodes can be of local, proxy, or *ghost* types. According to [74], a topological entity is classified based on its direct neighborhood. The direct neighborhood of an element is composed by all nodes (and other entities)

adjacent to the element, while the direct neighborhood of a node is composed of all elements (and other entities) indicent to it.

In summary, a communication layer's entity may be classified as three types:

- *Local* entity: When its direct neighborhood, in relation to the original mesh, is completely represented in the current partition, and the entity belongs to the current mesh partition (i.e., does not belong to the communication layer). Local entities include bulk elements, cohesive elements, and nodes.
- *Proxy* entity: When its direct neighborhood, in relation to the original mesh, is completely represented in the current partition, and the entity belongs to a remote mesh partition (i.e., belongs to the communication layer of its current partition). Proxy entities include bulk elements, cohesive elements, and nodes.
- *Ghost* entity: When its direct neighborhood, in relation to the original mesh, is *not* completely represented in the current partition. Ghost entities include only nodes.

Figure 5.2 illustrates a distributed representation of a 2-dimensional mesh for simplification motives. In (a), the initial mesh is shown, and (b) shows the partitioned mesh without the communication layer; (c) and (d) illustrate the mesh with the constructed layer. Black nodes are local nodes, while red nodes are proxy nodes that reference to local nodes on owner partitions. Blue nodes are ghost nodes that also reference local nodes on owner partitions. Gray elements indicate proxy elements that reference to local elements on owner partitions. Figure 5.3 illustrates the same logic in three-dimensions.

All proxy entities are processed like the local entities in relation to analysis attributes, such as mass and stress calculation. Ghost nodes are not processed the same way. They are synchronized by message passing of such attributes, as we will discuss later.

Given the following definitions, we can introduce the term "symmetric operations". Proxy nodes can be processed by gather or scatter algorthms. When processed by gathering technique, operations between its local and remote copy must be the same, i.e., they must have the same reference (adjacent) element and must be traversed exactly the same way (see discussion on randomness on GPU on Chapter 4). In our distributed simulation, however, we guarantee the operations are symmetric because we use coloring and the scatter technique. When constructing the communication layer, its coloring is exactly the same as its local copy, making the operations symmetric.

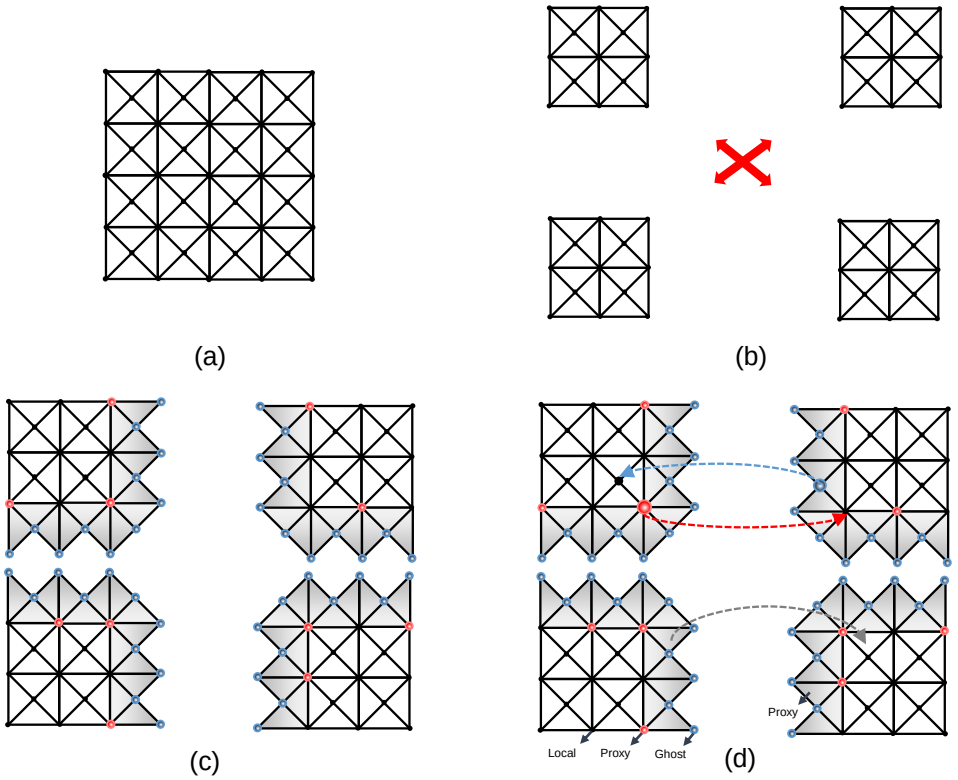

Figure 5.2 – 2-dimensional mesh partitioning. (a) Initial mesh; (b) Mesh partitioned and (c) with the communication layer and proxy and ghost entities. (d) illustrates each proxy and ghost entity refering to their local entity.

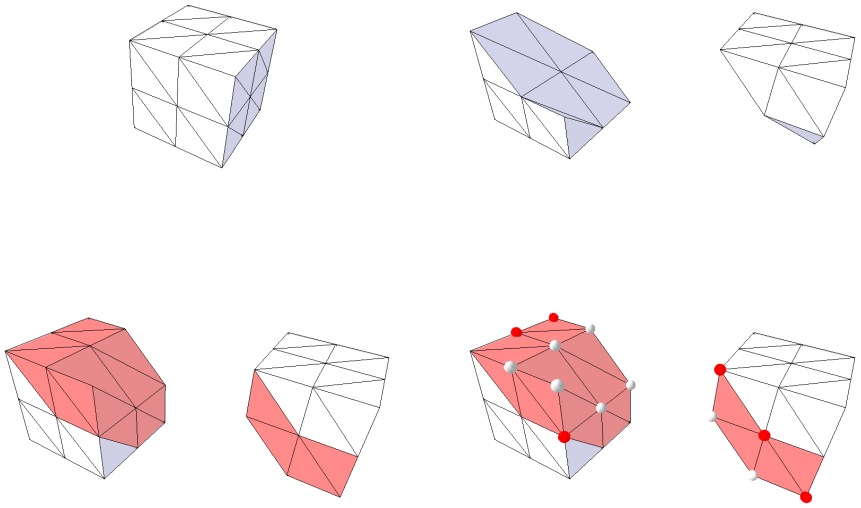

Figure 5.3 – 3-dimensional mesh partitioning. (a) Intial mesh. (b) Mesh partitioned and (c) with the communication layer. (d) Illustration of proxy (in red) and ghost (in white) nodes and proxy elements in red.

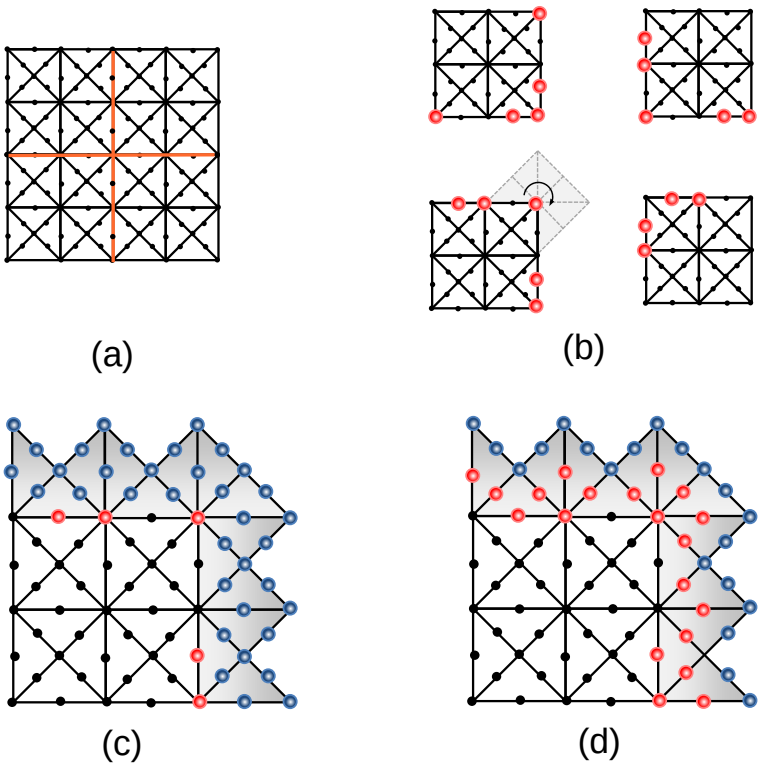

Figure 5.4 – Construction of the communication layer of a T6 mesh. (a) shows the initial mesh. (b) illustrates the partitioned mesh without the communication layer, but with the already classified local and proxy nodes. Adjacent elements from other partitions belonging to border nodes are swept and added to the communication layer (c) and (d) ghost nodes are classified according to facets.

5.2 Construction of the communication layer

After the original mesh is partitioned and proxy and local nodes are classified, proxy elements are added to the communication layer. This is done by going through all of the border nodes of the current partition and adding adjacent elements that belong to other partitions. We then add all nodes belonging to the new elements. These nodes are initially categorized as ghost nodes. After updating element adjacency, we verify for each node if their direct neighborhood is completely represented (i.e., if contains all adjacent elements from the original mesh). If so, we set them as proxy node; else, we keep them as ghost. Figure 5.4 shows a two-dimensional example.

5.3 Data Structure

In the previous section, we explained how the distributed mesh and communication layer are represented and built. Now, we describe implementation details of the data structure used to insert cohesive elements and synchronize

topological and analysis attributes. We extend the proposed data structure in Section 3.1 to 3D and include information that will help distributed mesh representation.

Before explaining the extension of the 3-dimensional data structure, it is worth noting some important attributes we need to store that will help us keep track of mesh topology synchronization. Each node and element belong to one owner partition and have an unique entity identifier inside that partition. Therefore, we must store for all remote proxy and ghost (remote) and local (owner) entities the identifier of the owner partition it is contained in and the owner entity identifier inside that partition. While the partition *id* will be used to send communication messages to remote partitions, the owner *id* is used to access the owner entity inside the owner partition. As explained in Section 5.1, topology entities can be classified as local, proxy, or ghost. We also need to store this information in the topology table in the data structure to easily identify the communication layer and at the same time without costing too much device memory.

One of the major steps of the synchronization strategy that will be greatly used in our simulation is accessing a reference (adjacent) element of a given node when sending a message. To fetch its new owner *id*, a ghost node may have been duplicated in the other partition and therefore there is no clue which is the owner node. To access the owner node, we use one of the remote node's adjacent element since bulk elements remain unchanged in the mesh. A tuple *(owner entity handle, local id)* message is sent to the owner partition (where *local id* is the incidence of the node in that element), and the owner element is accessed inside the owner partition. From that element, we use the *local id* to access the owner or proxy node. Details of synchronization are discussed later in the chapter. The adjacent element plays a key role in messaging and without it we would not be able to synchronize the communication layer. Therefore, we must store, for each node in the node table, one reference (adjacent) element to it.

Given that we are now in three-dimensional situation, we represent bulk elements as tetrahedron elements (Tet4) and cohesive elements as linear triangular elements (T3). Figure 5.5 shows the 3D element types: node, bulk element, and cohesive element. Because we are now in 3D space, memory consumption will be a lot bigger and the node traversal path won't be restricted to a two-way algorithm. Instead, we now face a challenge to go through a series of elements in a graph-like structure, which requires a stack to traverse the elements. Therefore, a simple and low-memory consumption data structure is key to implement such algorithms. We will later come back to this point when

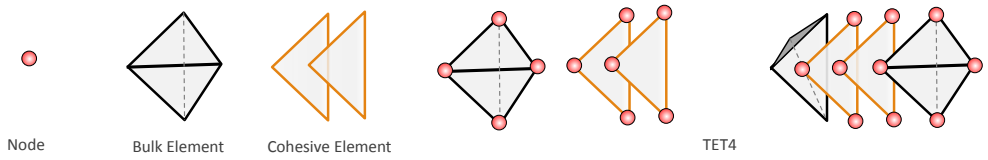

Figure 5.5 – Element types used in the 3D analysis code. Bulk elements are volumetric tetrahedron elements and cohesive elements are linear triangular elements.

we discuss the insertion of cohesive elements. Another important point is that the number of colors in the mesh greatly increases as the valence of the nodes increases, so more kernels must be launched. Furthermore, the 3D analysis code is much more complex than the 2D one, requiring more global memory accesses, more register use, and more numerical operations. All this makes the 3D simulation much more challenging to implement in the GPU.

We propose an extension of the data structure similar to the 2D presented in Section 3.1, plus additional attributes that will help with the distributed simulation. Figure 5.6 illustrates our proposed data structure. We have the node and element table with node positions and element node references and opposite elements like in two-dimensions. In the node table, we add a z component. Because we are now using tetrahedron elements for bulk elements and linear triangular elements for cohesive elements, we use four nodes referenced in the node table for bulk element nodes and six nodes for cohesive elements. Four element indices are used to access each bulk element node's opposite elements and two indices are used for cohesive elements (one for each facet). Much like the adaptive simulation in 2 dimensions, we store, for each node, one adjacent element. However, in the adaptive case, while using the element for means of node traversal, we now use it for messaging purposes, as explained above. The node table is completed with the identifier of the owner partition of that node and the identifier of the local node *id* inside the owner partition. We indicate if a node is local, proxy, or ghost by reserving two bits in the partition *id*. The element table is completed similarly to the node table. The identifier of the owner partition and the element inside the owner partition are set in the table. A proxy bit is reserved in the owner partition identifier to indicate if the element is proxy.

With the extended 3D data structure, we are able to traverse the node's incident elements just like the two dimensional case. The traversal is like a graph structure in which it is suspended when the adjacent element is reached again or all elements are visited. This uses a lot of shared memory, as we will see later in the chapter. We are also able to send messages to remote partition and receive incoming topology responses and data attributes such as stress, velocities, and

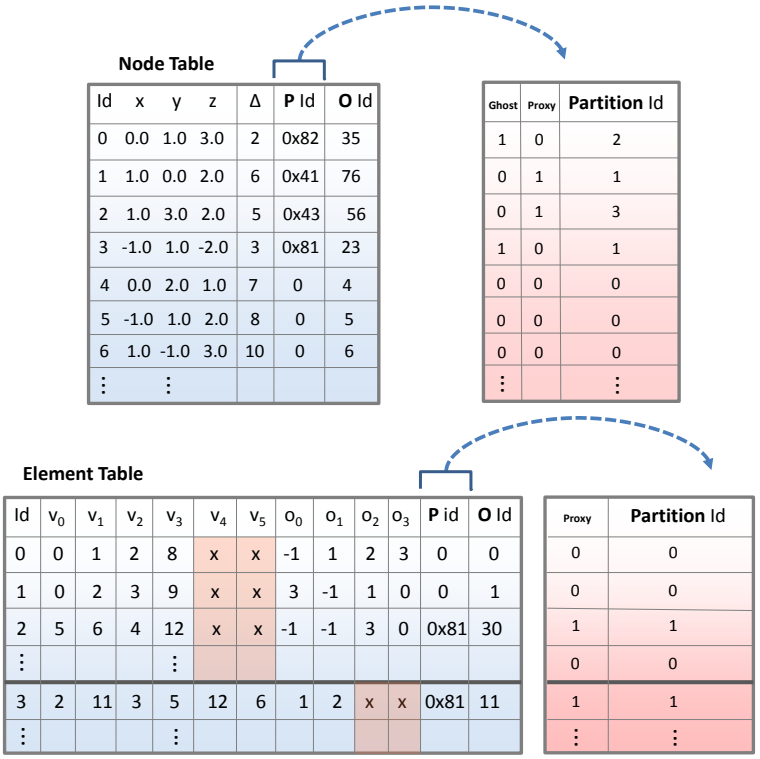

Figure 5.6 – Data strucuture for distributed 3-dimensional finite element analysis. The node table contains the positions, one adjacent element *id* belonging to its incidence, and the owner partition and entity *id* it belongs to. If the owner patition is the one it is in, the owner *id* is the same as the node table. The partition *id* contains 2 bits that indicate if the node is local, proxy, or ghost. The remaining bits refer to the owner partition identifier. The element table contains six nodes, in which four are used for tetrahedron bulk elements and six are used for triangular cohesive elements. Four opposite identifiers indicate elements opposite to the bulk elements' four nodes and the cohesive element's two facets. The owner partition *id* reserves one bit to indicate if the element is local or proxy and the remaining bits to indicate the partition of the owner. The owner identifier of the entity follows next.

accelerations. To do this, an acess is made to the adjacent element stored for a given node. The owner partition of node is fetched from the node table and the message is sent to it containing the tuple *(owner* $element_{adj}$ *handle, local id)*, where owner $element_{adj}$ handle is fetched from the element table and the *local id* is the incidence of the node in that element. Notice that the *local id* is the same for all respective local and their remote entities (bulk and cohesive elements). The message is sent to the owner partition of the element and used to access the node of that element using *local id*. The adjacent element in turn sends a reply message with the response data containing the *id* of the node in that partition to the sender using the same procedure. Notice that this node can also be of a proxy category. Figure 5.7 illustrates the process.

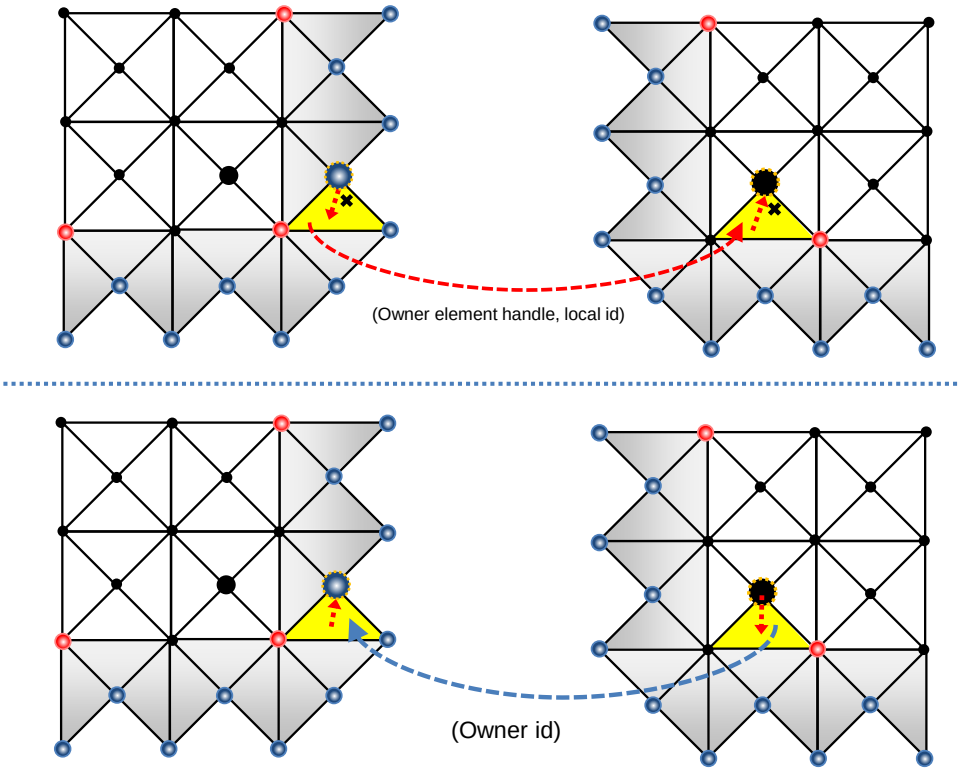

Figure 5.7 – Messaging Procedure to obtain topology or analysis data. (1) The node accesses the adjacent element and sends a message to its owner with the tuple *(owner element handle, local id)*. (2) The owner of the element receives the message and accesses the node with it. (3) The receiver responds the message with the topology or analysis attribute requested by the sender.

5.4 Insertion of cohesive elements

Like the two-dimensional simulation, after fractured facets are extracted from the mesh, we filter the elements that contain them and launch a kernel per color to insert cohesive elements. Again, we choose the dominant element that will take over the fractured facet and will duplicate the node. Our criteria is the bulk element with the least *id*. With our extended data structure, we are able to insert cohesive elements in volumetric tetrahedron elements by traversing the nodes just like the triangular meshes. The 2-dimensional case, however, was restricted to a two-way element traversal. While this simplifies the GPU code, we cannot perform this in the three-dimensional case. Instead, there are several paths starting from an element of a given node that leads to the adjacent element, if not blocked by a cohesive element. Therefore, we need a stack to keep track of the visited elements. We use the breadth-first search algorithm in the GPU. The stack is kept in shared memory, thus the amount of memory is determined by the maximum node valence. In 2D, we duplicated the node only if the adjacent element to the fractured facet is not reached in the

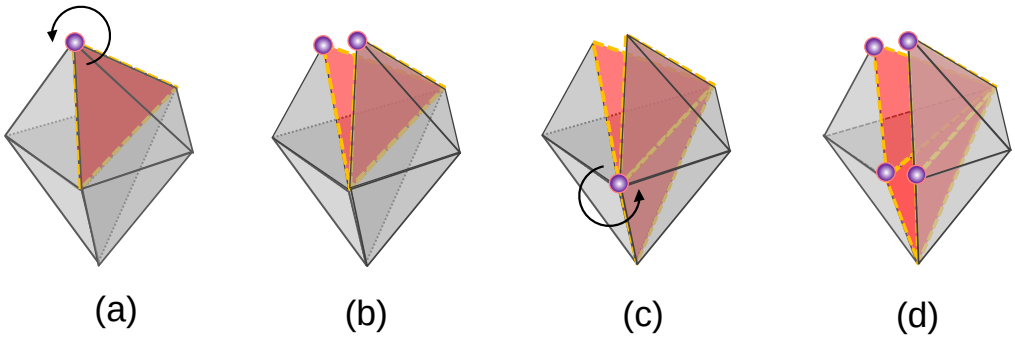

Figure 5.8 – Insertion of cohesive elements in 3D. Node is traversed using a Breadth-first search in shared memory. If there is at minimum one path to the adjacent element, the node is not duplicated.

current traversal. However, we now have multiple paths to reach the adjacent element. Therefore, if there is at least one path leading to the adjacent element, the node is not duplicated. If the adjacent element is not reached (i.e., the path is blocked by a cohesive element or mesh boundary), we duplicate the node. Figure 5.8 illustrates inserting cohesive elements in volumetric tetraheda elements through breadth-first search node traversal.

When creating a new node, an adjacent element must be assigned to it. It is of extreme relevance which type of bulk element (i.e. local or proxy) we assign the node. Therefore, for local and proxy nodes, we assign the same bulk element for each partition by choosing the one with the least owner partition *id* and least owner element *id*. This will avoid creating inconsistencies in the mesh, such as proxy nodes having multiple local copies in other partitions. The ghost nodes case is different because they may not have all the elements of their proxy/local copies, so any adjacent element may be chosen.

Espinha et al's [74] strategy to insert cohesive elements and duplicate nodes in a distributed mesh is based on duplicating local and proxy nodes and inserting local and proxy cohesive elements, synchronizing proxy entities based on reference to owner entities, and finally updating ghost nodes based on owner *ids*. In the third step, ghost nodes are updated by duplicating them when needed and updating reference to owner nodes. First, local elements which are incident to duplicated nodes and have corresponding proxy element in other partition are selected to send message to their remote partitions about any potential node duplication that would affect ghost nodes. Remote partitions receive the message and send a request to the owner partition of the element, which receives the request and sends the message with the new owner *id*. To map this to the GPU, we would face two challenges that would greatly reduce the performance of our simulation. First, it would require to traverse the nodes twice: when filtering proxy elements from other partitions and when updating

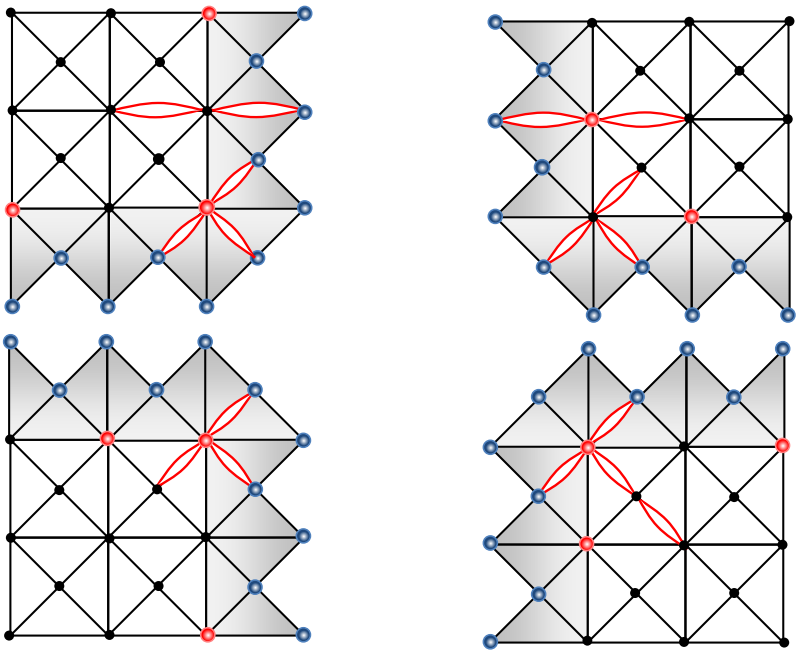

Figure 5.9 – Phase 1 of inserting cohesive elements in partitioned meshes. Local and proxy cohesive elements are inserted.

ghost nodes with their new owner *ids*. Second, this algorithm generates "holes" in the node table, thus requiring extra processing either to collapse or to reuse the nodes.

We propose a different strategy based on duplicating ghost nodes during phase one of inserting local and proxy cohesive elements. Figure 5.9 shows the initial 2D mesh (for simplification purposes) with local and proxy cohesive elements inserted. Using the traversal algorithm to duplicate, we duplicate all nodes in the mesh when needed, *including ghost nodes*. Notice that the ghost node of the remote partition may have been duplicated while its correpondent owner node may not, which is fine. Figure 5.10 shows the nodes after the duplication and insertion of cohesive elements. With ghost node duplication, there is no need to synchronize ghost nodes owner *ids*, since synchronization is done via element messaging. This greatly reduces the communication bottleneck and avoids node traversal algorithms and node reuse that would need to be treated in the GPU. It also eliminates phase three of Espinha et al's insertion of cohesvie elements algorithm when updating ghost nodes.

Phase two of insertion deals with synchronizing proxy entities *ids*, or updating entity references to owners in local partitions. Phase two updates proxy nodes and proxy cohesive elements owner ids to their owners. Figure 5.11 illustrates the procedure to update proxy entity owner attributes. If the entity is a node, the reference element is used to send the message to its owner

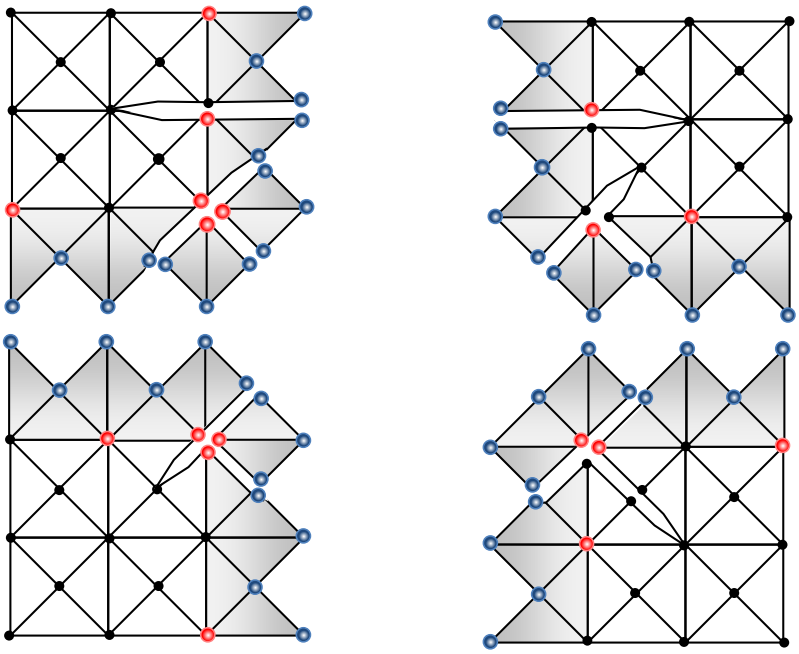

Figure 5.10 – Phase 1 of insertion of cohesive elements in partitioned meshes. Local, proxy, and ghost nodes are duplicated as a result of the cohesive elements insertion. Adjacent elements are updated accordingly.

using the tuple *(owner element handle, local id)*. It is important to remember this reference (adjacent) element must be proxy since the node is also proxy. If the entity is a cohesive element, we get one of its proxy adjacent element and send the message using the tuple *(owner element handle, local id)*. We do this only for new proxy nodes and cohesive elements. When the message gets to the owner partition, the owner element receives it and accesses the node using the *local id*. The element then responds with its owner id and the remote partition updates the proxy node with it.

5.5 Parallel simulation

This section deals with the cohesive fracture and fragmentation simulation of distributed three-dimensional meshes. The 3D simulation procedure is shown in Table 2.1. In a pre-processing phase, the stiffness matrix is computed for each elements and the nodal mass is updated for each node. In the simulation loop, displacements are updated from velocities and accelerations. Next, nodal stress is computed from element gauss points and, for each facet, if three of its nodal facet's stress exceeds a threshold, we indicate the facet as fractured. Next, cohesive elements are inserted, followed by and update on nodal masses. Internal and cohesive forces are computed and used to calculate velocities and

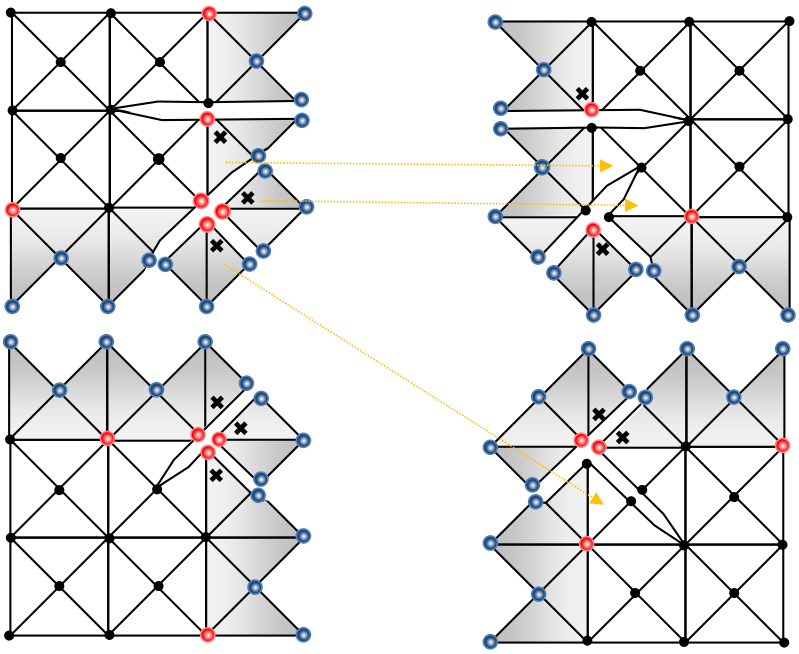

Figure 5.11 – Phase 2 of insertion of cohesive elements. References of proxy nodes and cohesive elements to their owners are updated by sending message via their adjacent elements.

accelerations. Boundary conditions are then applied in the model. Notice that the simulation is a lot costly due to the fact that there is a degree of freedom more. Stress matrices that were 2×2 are now 3×3, for example. However, internal force matrices remain 12×12. An increase in arithmetic operations in cohesive and stress *kernels* leads to an increase in GPU register use and greater GPU memory consumption leads to an increase in global memory access and consumption.

Table 5.1 shows the distributed 3D simulation procedure with data and topology synchronization stages. After computing nodal stresses, ghost partitions have their stress outdated because they have undefined neighborhood. Therefore, it is necessary to synchrionize stress among all ghost nodes in respect to their owner/proxy nodes. After stresses are updated in the communication layer, cohesive elements are inserted accordingly and nodal masses updated. At this point, ghost nodes may have their respective remote entities' adjacency changed, so we must synchronize nodal masses and adjacencies. Any proxy entities that are now created must be updated to their respected owner entities (i.e., nodes and cohesive elements), as explained in Section 5.4. After internal and cohesive forces are used to calculate velocities and accelerations, boundary conditions are applied to the model. We then must synchronize velocities and accelerations in ghost nodes. A boundary condition may fall exactly in a

communication layer and belong to two partitions, so synchronizing velocities and accelerations after updating them is very important, especially when they are used to calculate the displacements in the next timestep.

```
1: Compute Stiffness Matrix
2: Update Nodal Mass
3: current step ← 0
4: while current step <= maximum step do
5:    Update Displacements
6:    if current step == check step then
7:       Compute Stresses
8:       Synchronize Nodal Stresses
9:       if stresses > stress threshold then
10:         Insert Cohesive Elements
11:         Update Nodal Masses
12:         Synchronize Node Adjacency and Masses
13:         Synchronize Proxy Entities
14:      end if
15:   end if
16:   Compute Internal Forces
17:   Compute Cohesive Forces
18:   Update Velocities and Accelerations
19:   Update Boundary Conditions
20:   Synchronize Nodal Velocities and Accelerations
21:   current step + = 1
22: end while
```

Table 5.1 – 3D distributed fracture and fragmentation algorithm

The ghost nodes synchronization stage is done by accessing its adjacent elements rather than sending message directly to its owner node *ids*, since we do not have that information (which was ceased to be calculated in the topology synchronization stage). By fetching one of its adjacent elements stored in the data structure, a tuple message *(owner element$_{adj}$ handle, local id)* is sent to its owner partitions, like the algorithm described in Section 5.4. At the owner partition side, the owner node is accessed and the data (i.e., stress, velocities and accelerations) are fetched and sent back to the remitent. The ghost nodes are then updated.

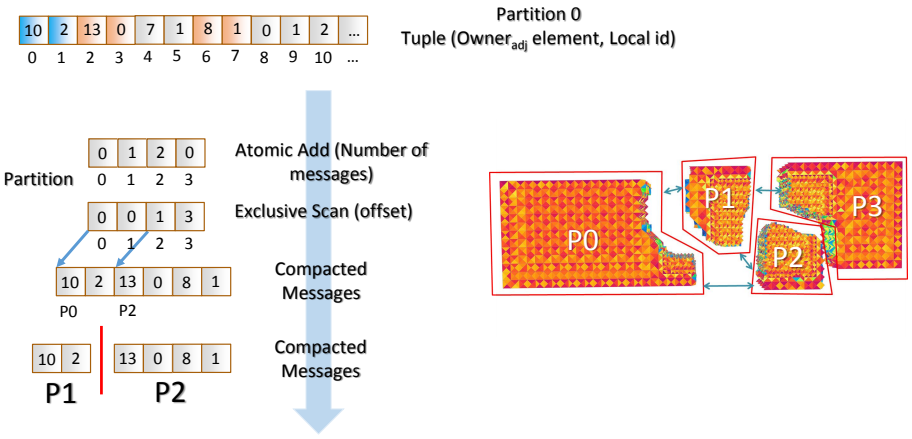

Figure 5.12 – This figure shows the compacting and sending of messages to neighbor partitions. First, ghost nodes of the local partition are swept and count how many messages it must send to each partition via atomic intrinsincs. Next, we scan the number of messages per partitions to create an offset array to fill the messages. We sweep the nodes again and using the offset array and the "number of messages per partition" array, we fill the tuple messages per partition and copy them to the CPU.

5.6 Message extraction and sending

To synchronize topological attributes from proxy nodes and especially analysis attributes from ghost nodes (stress, displacements, masses...), messages must be sent to neighbor partitions (i.e. to corresponding local or proxy nodes). To build the message, the following steps are done, as illustrated in Figure 5.12. For simplification purposes, our discussion deals with ghost node synchronization.

First, we launch a kernel with a thread per ghost node. Each thread accesses the reference element from the node and adds one to a counter of the respective bulk element owner partition. This results in a device array with the number of messages that will be sent per partition. Next, we do a prefixed scan on the array, which gives us the offset that each partition will need to start writing each tuple to the message array. The next kernel launches one thread per ghost node and fills the array by writing the tuples *(owner element handle, local id)* for each partition, using the offset array and incrementing the node counter for each tuple. The message and the "number of tuples per partition" array are copied to the CPU and the message is sent to each partition.

One step of our simulation is copying excessively data from the *host* to *device* and from *device* to *host*. Whenever a message is sent from a given compute node, the GPU processes and extracts that message and copies it to the CPU so that *MPI*[1] sends it via network. The receiver in turn receives

[1]Message Passing Interface

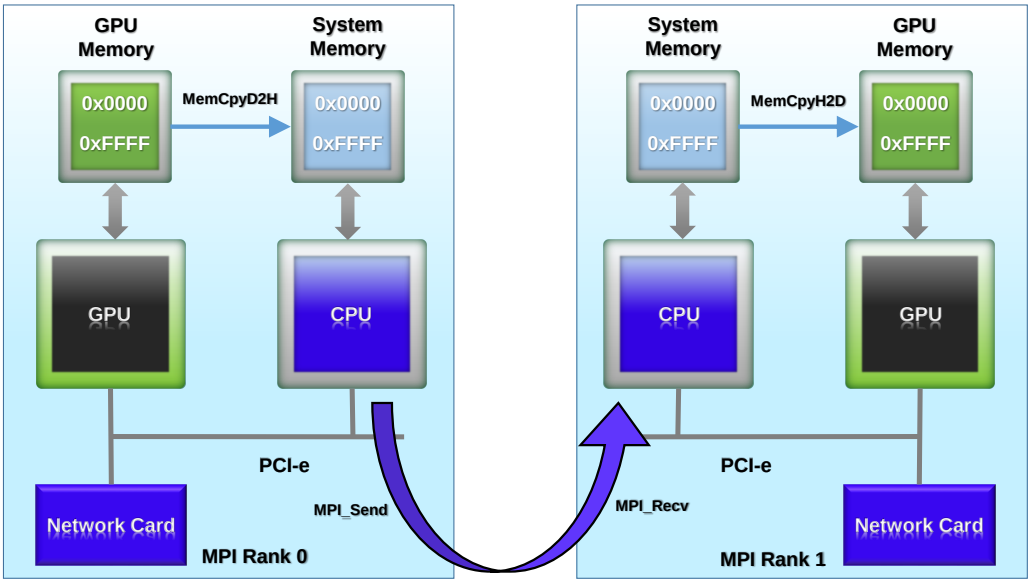

Figure 5.13 – This figure illustrates the message trasfer between two computer nodes with GPUs. It is a costly stage of the simulation due to the fact of the memory trasfer between CPU and GPU before sending messages to other computer nodes and after receiving messages.

the message and copies the data to the GPU for further processing. This is undoubtedly one costly step of our program due to network bottleneck and done at every timestep, both for stress messages, node masses and adjacency, and velocity and acceleration messages. Figure 5.13 illustrates the procedure.

5.7 Experimental results

In this Section, we ran a set of computational experiments regarding the 3-dimensional finite elements simulations on a single GPU card. The experiments were split in two parts: inserting cohesive elements decoupled from mechanics analysis and running the fracture and fragmentation simulation.

Insertion of cohesive elements

Like in Section 3.3.1, we ran a computation experiment to check the consistency of the insertion of cohesive elements decoupled from the mechanics analysis code. We used a ring specimen shown in Figure 5.14(a) and post-cohesive elements insertion shown in Figure 5.14(b). We set up experiments similar to the ones described by Pandolfi and Ortiz [63] and by Paulino et al. [61], as described in Section 3.3.1. Cohesive elements were inserted, in a random order, by grouping the facets in sets and inserting 5% of cohesive elements serially and concurrently within each facet group.

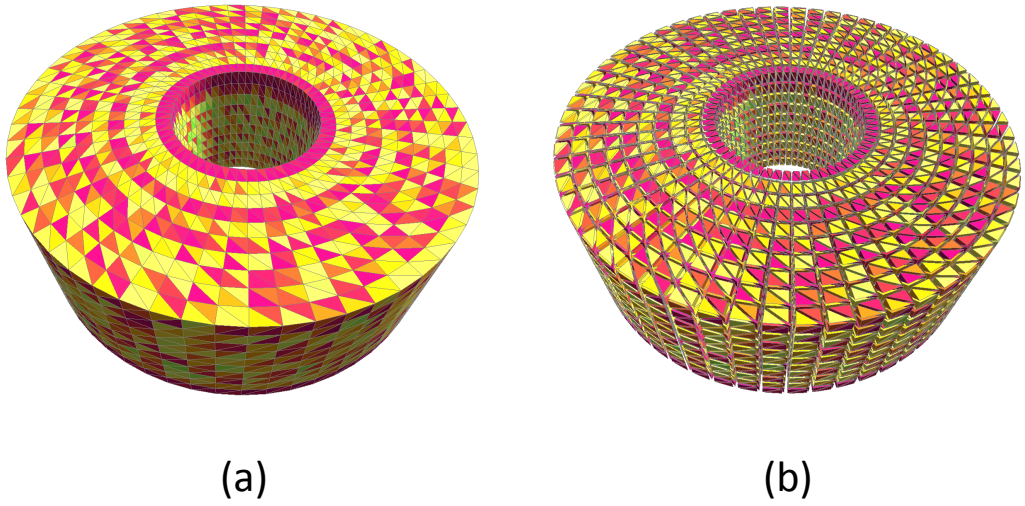

Figure 5.14 – Colored 3D ring specimen (a) and after the insertion of cohesive elements (b).

The GPU simulation results are compared to CPU counterparts running on a Intel Core i7 CPU @ 2.80GHz with 12GB of memory on a 64-bit Windows 7 operating system. The GPU used device is a NVIDIA GeForce GTX 480 with 15 multiprocessors, each with 32 cores and a total of 480 CUDA cores, with a clock rate of 1.40 GHz and using compute capability 2.0 because we use double precision in the simulation. The total amount of GPU memory is 1.536 Gigabytes.

We employed a Tet4 mesh like the one in Figure 5.14 and used the same greedy algorithm as in Section 3.3.1 to color the mesh. The number of colors obtained were 32. Because we currently run in 3D, the number of colors naturally increases as the node valence increases. The model was discretized up to 4,500,000 bulk elements and 780,300 nodes, reaching a maximum amount of 8,940,000 cohesive elements inserted in the largest mesh. Table 5.2 shows the results of cohesive elements insertion for different ring discretizations. We were able to validate the GPU results and topology consistency.

Bulk elements	**Initial nodes**	**Final nodes**	**Cohesive elements**	**CPU Time (s)**	**GPU Time (ms)**	**Speedup**	**Efficiency**
36,000	7,260	144,000	69,600	7.428	38.161	194.6	40.5 %
288,000	52,920	1,152,000	566,400	60.972	122.434	498.0	100.0 %
972,000	172,980	3,888,000	1,922,400	209.726	387.143	541.7	100.1 %
2,304,000	403,440	9,216,000	4,569,600	489.204	910.982	537.0	100.1 %
4,500,000	780,300	18,000,000	8,940,000	954.472	1,792.714	532.4	100.1 %

Table 5.2 – Results for 3D insertion of cohesive elements in Ring specimen, decoupled from analysis code.

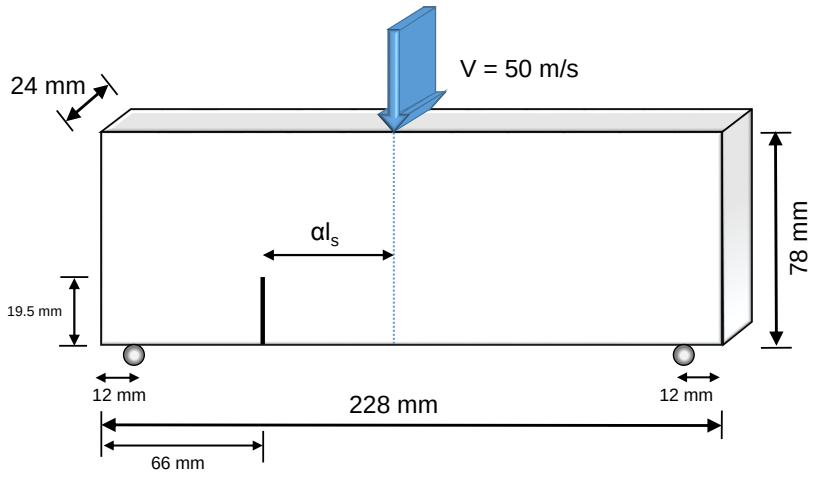

Figure 5.15 – 3D mixed-mode problem geometry and loading conditions.

Parallel simulation

We have tested the cohesive fracture simulation on a single GPU using the pre-cracked three-point-bend beam (3D beam) shown in Figure 5.15 and referred in the experiments of John and Shah [75]. The beam is 228 mm in x per 78 mm in y per 24 mm in z and has an initial notch starting at x = 66 mm and extends until y = 19.5 mm. Two supports conditions on positions x = 12 mm and x = 216 mm are fixed below the beam. An initial load of v = 50 m/s is applied on top of the model at a distance d $= \alpha l_s$, where l_s is the distance between the two support conditions.

The GPU simulation result for the coarse mesh is compared to CPU counterparts running on a Intel Core i7 CPU @ 3.40GHz with 32GB of memory on a 64-bit Windows 7 operating system. The GPU used device is a NVIDIA GeForce GTX Titan with 2688 CUDA cores using compute capability 2.0 because we use double precision in the simulation. The total amount of GPU memory is 6 Gigabytes.

We have tested the 3D simulation on a coarse version of the mixed-mode beam and on its refined version. The coarse version contains 113,984 bulk elements and 25,777 nodes. The refined version contains 716,736 bulk elements and 137,490 nodes. It is imperative to notice that the refined version pushes the limits of the GPU in terms of memory usage. We employed Tet4 (tetrahedra) elements on the mesh and T3 cohesive elements. The greedy algorithm was used to color the mesh and the number of colors obtained were 55 on the coarse mesh and 65 on the refined mesh. The mesh is initially refined in the middle left where the fracture will tends to propagate. Cohesive elements were checked for insertion at each 10 steps of the simulation. The coarse mesh had a time step of 30 ns and 9,000 steps, while the refined version's simulation ran with 73,000

steps and a timestep of 3 ns. Initial material parameters are: initial velocity = 50 m/s, elastic modulus = 29 GPa, Poisson coefficient = 0.24, specific mass = 2400 kg/m3. Fracture energy materials are as follows: fracture energy G_I = 22 N/m, cohesive strength smax = 8 MPa, and shape parameter $\alpha = 2$.

Figure 5.16 shows moments of the simulation of the coarse mesh at certain timesteps. Before the crack growth begins, energy concentrates on the load in the middle of the beam at t = 100 μs and along the support conditions at t = 170 μs. The crack initiates from existing crack tip at about t = 180 μs, and numerous fragmentations occur around the main crack path. The main crack clearly propagates at an angle of about 30°. Figure 5.17 shows the refined version of the 3D mixed-mode beam with 716,736 bulk elements at step 73,000. The crack propagated as expected like the coarse mesh did and both results were consistent with the experiments of John and Shah [75].

Tables 5.3 and 5.4 show analysis attributes and performance results for both simulations. We were able to run the coarse mesh on the CPU and compared to the GPU counterpart. The GPU time was 24.59 s, while the CPU version of the same mesh was 706.74 s. The refined version took longer since the timestep is ten times smaller and the number of steps is approximately 8 times greater. It took 906.62 s on a GeForce GTX Titan.

Mesh type	Bulk elements	Nodes	New nodes	CHZ elements	Colors
3D Beam 38x13x4	113,984	25,777	1,079	2,728	55
3D Beam 76x26x8	716,736	137,490	2,010	5,886	65

Table 5.3 – Simulation and mesh parameters for mixed-mode 3D beam mesh and its refined version.

Mesh type	Timestep	Steps	CPU time	GPU time	Speedup	Efficiency
3D Beam 38x13x4	3.0e-8 s	9,000	706.74 s	24.59 s	28.74	1.1 %
3D Beam 76x26x8	3.0e-9 s	73,000	-	906.62 s	-	-

Table 5.4 – Simulation and mesh parameters and results (GPU and CPU time) for mixed-mode 3D beam and its refined version.

Figure 5.18 presents results for average kernel times. Like the results presented in Section 3.3.2, the stress kernel dominates the average time graph due to its excessive numeric operations for a GPU kernel, using a great amount of registers. Even with the strategy of splitting kernels, it still occupies a great portion of the simulation. However, because it is executed at each 10 steps of the simulation, the kernel that continues dominating the total simulation time is the internal force kernel, due to its global memory accesses. Even optimizing it by using texture memory, shared memory, and thread distribution on matrix

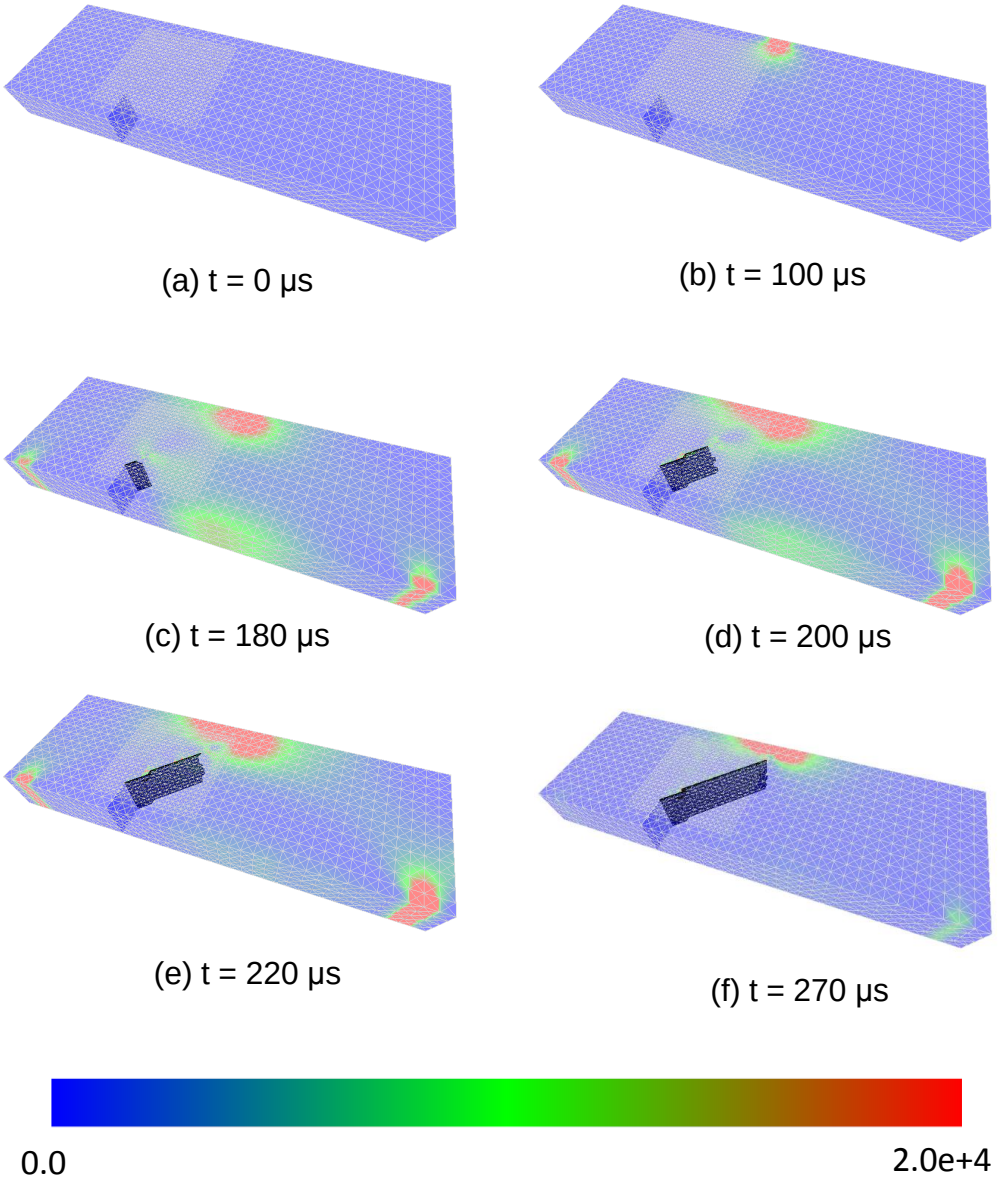

Figure 5.16 – Strain contour and crack propagation plots of the 3D mixed-mode experiment at different time instants: **a** 0 μs; **b** 100 μs; **c** 130 μs; **d** 180 μs; **e** 200 μs; **f** 270 μs

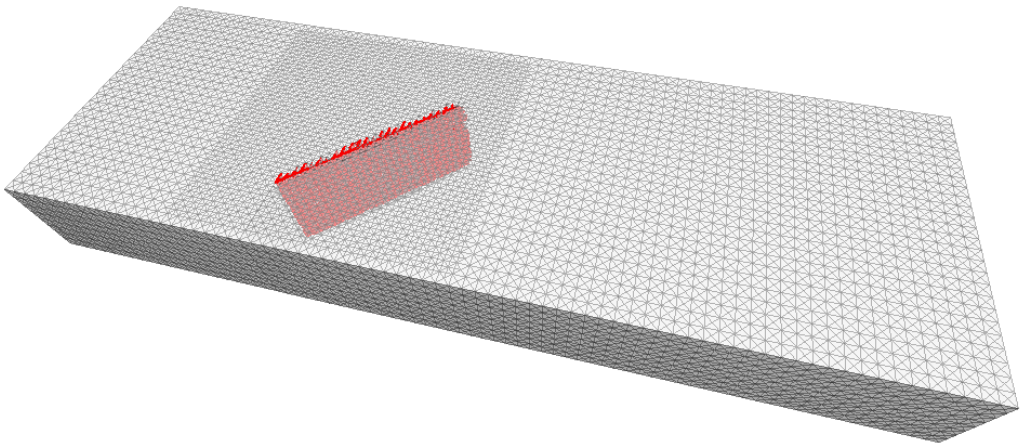

Figure 5.17 – Crack propagation plots of the refined version 3D mixed-mode experiment with 716,736 bulk elements, at time instant t = 270 μs.

lines, it still dominates the simulation. An average time increase of kernels like node duplication and cohesive kernels from the 2D simulation is also noticeble due to the 3D simulation extension.

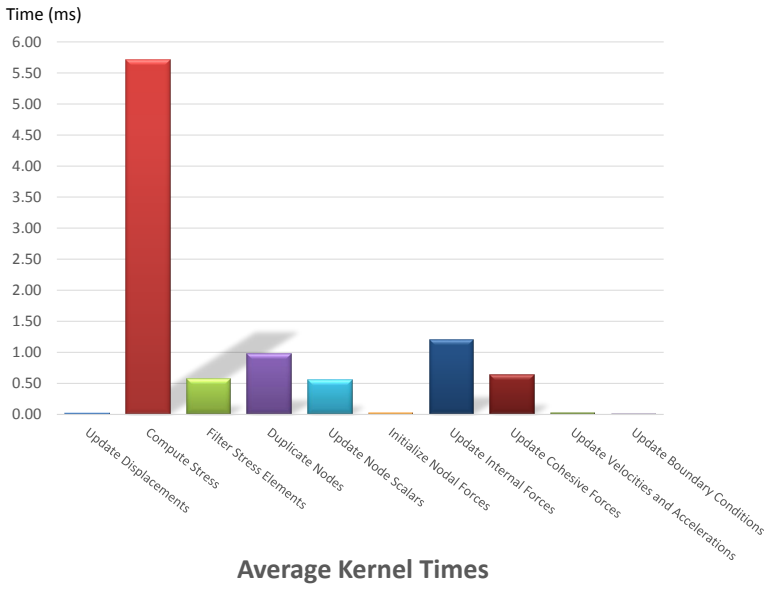

Figure 5.18 – Average time of each *kernel* of the simulation for a Tet4 mixed-mode 3D beam mesh with 113,984 bulk elements.

5.7.1
3D Distributed fragmentation simulation

In this section, we simulate the 3D analysis code using distributed cloud computer clusters, each containing a GPU, which process our model and send

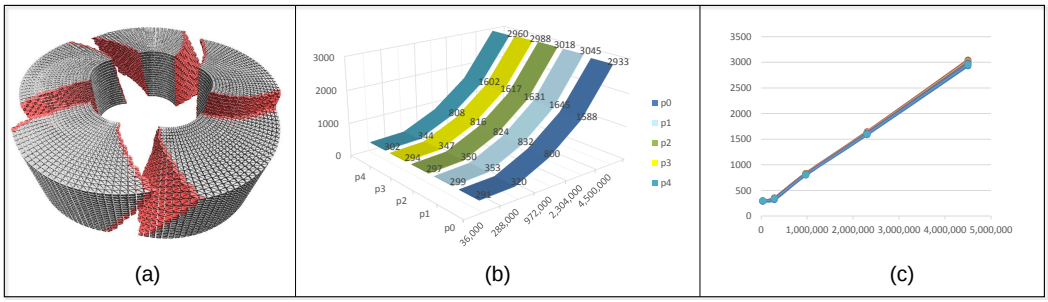

Figure 5.19 – (a) 3D Ring model with 5 partitions after the cohesive elements insertion. (b) and (c) Cohesive insertion-time graph for each partition and discretization.

messages to communicate with other partitions. We have employed four versions the 3D mixed-mode beam used in Section 5.7: a reduced-scale specimen and a mid-scale specimen shown in Figure 5.20; and two full-scale specimens of different dimensions to test the simulation using a large number of GPU nodes.

Insertion of cohesive elements

To test the cohesive elements insertion, we used one CPU node using with up to 8 threads to verify the corretude of the algorithm and the synchronization step (i.e., synchronizing proxy and ghost entities). We used the ring model to test the linearity of the insertion time. Figure 5.19(a) shows the ring model divided into five partitions with its communication layers in red and after inserting all cohesive elements. We varied the ring model in five discretizations from 36,000 bulk elements to 4.5 million bulk elements. We followed the experiments of Pandolfi and Ortiz [63] and Paulino et al. [61], where cohesive elements were inserted in random order and grouped into 5% of the mesh's facets. Figure 5.19(b) shows a graph for each partition, varying the discretization outputing the time. As we can see by 5.19(c), there is a linear behaviour obtained by the insertion algorithm.

Distributed simulation

To perform the distributed analysis simulation, we used Amazon Web Services[2] clusters from their cloud. Each compute node is composed of an Intel Xeon E5-2670 Processor with 15 GB of memory and an NVIDIA GRID K520 CUDA capable device with CUDA version 6.5, 4 GB of total amount of global memory, and 8 Multiprocessors with 192 CUDA cores each (total of 1536 CUDA Cores). We use OpenMPI 1.8.4 as Message Passage Interface [3] for communication between the nodes. To partition the mesh, we used METIS, a

[2]http://aws.amazon.com/
[3]http://www.open-mpi.org/

serial graph partitioning program[4].

We first tested the reduced-scale and mid-scale models in one CPU node and varying the number of partitions. Tests were made in order to assure the consistency of the result, both topological and analytic. For example, to ensure topological consistency of the mesh, we verify if all proxy nodes point to the same proxy adjacent elements; if a local node is unique among all mesh partitions; if the number of adjacent elements in local and proxy nodes are exactly the same in all partitions; if ghost nodes point to proxy adjacent elements; if node owner is equal to local node handle by accessing its adjacent element and local id.

Regarding the distributed simulation with GPUs, we first tested reduced-scale like-version of the mixed-mode beam of Section 5.7 and shown in Figure 5.20, varying the number of compute nodes from one through five. Next, the mid-scale specimen was tested varying the number of compute nodes from one through five, folowed by the two full-scale specimens, each running in 15 compute nodes. The coarse version contains 113,984 bulk elements and 25,777 nodes. The mid-scale version contained 716,736 bulk elements and 137,490 nodes. The full scale specimens contained 2,351,424 bulk elements and 445,658 nodes, and 5,495,168 bulk elements and 1,035,078 nodes, respectively. The greedy algorithm was used to color the mesh and the number of colors obtained was 55 for the reduced-scale specimen; 65 for the mid-scale specimen; and 63 for both full-scale specimens. The mesh is initially refined in the middle left where the fracture will tend to propagate. Cohesive elements were checked for insertion at each 10 steps of the simulation for all models. The coarse mesh had a time step of 30 ns and 9,000 steps; the mid-scale had a time step of 3 ns and 73,000 steps; the first discretization of the full-scale specimen had a time step of 3 ns and 73,000 steps; and the second discretization of the full-scale specimen had a time step of 1 ns and 220,000 steps. Initial material parameters are, according to [61]: initial velocity = 50 m/s, elastic modulus = 29 GPa, Poisson coefficient = 0.24, specific mass = 2400 kg/m3. Fracture energy materials are as follows: fracture energy G_I = 22 N/m, cohesive strength smax = 8 MPa, and shape parameter $\alpha = 2$.

[4]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview/

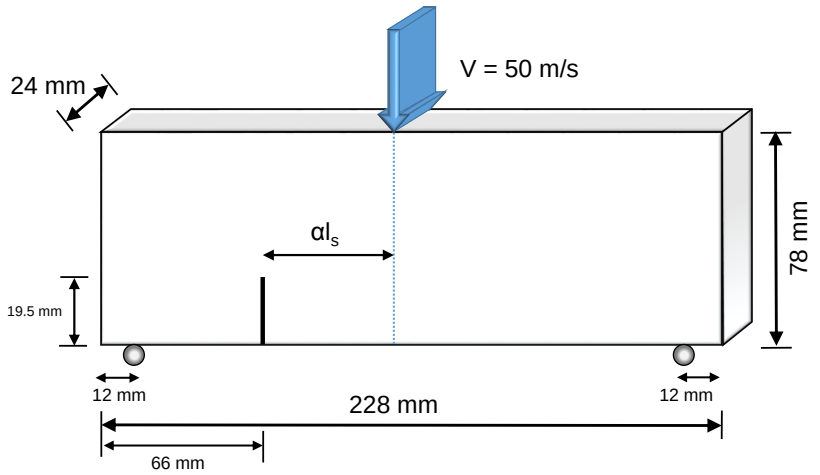

Figure 5.20 – Distributed 3D mixed-mode problem geometry and loading conditions.

3D Beam 38x13x4							
CPUs	**Nodes**	**B. Elements**	**Colors**	**Timestep**	**Steps**	**Time**	**Efficiency**
1	25,777	113,984	55	3.0e-8 s	9,000	41.6 s	–
2	25,777	113,984	55	3.0e-8 s	9,000	35.5 s	58.5%
3	25,777	113,984	55	3.0e-8 s	9,000	32.5 s	42.7%
4	25,777	113,984	55	3.0e-8 s	9,000	29.2 s	35.5%
5	25,777	113,984	55	3.0e-8 s	9,000	28.9 s	28.8%
3D Beam 76x26x8							
CPUs	**Nodes**	**B. Elements**	**Colors**	**Timestep**	**Steps**	**Time**	**Efficiency**
1	137,490	716,736	65	3.0e-9 s	73,000	33.0 min	–
2	137,490	716,736	65	3.0e-9 s	73,000	19.4 min	85%
3	137,490	716,736	65	3.0e-9 s	73,000	14.4 min	76.3%
4	137,490	716,736	65	3.0e-9 s	73,000	11.9 min	69.3%
5	137,490	716,736	65	3.0e-9 s	73,000	9.6 min	68.8%
3D Beam 114x39x12							
CPUs	**Nodes**	**B. Elements**	**Colors**	**Timestep**	**Steps**	**Time**	**Efficiency**
15	445,658	2,351,424	63	3.0e-9 s	73,000	15.0 min	–
3D Beam 152x52x16							
CPUs	**Nodes**	**B. Elements**	**Colors**	**Timestep**	**Steps**	**Time**	**Efficiency**
15	1,035,078	5,495,168	63	1.0e-9 s	220,000	82.5 min	–

Table 5.5 – Topology and simulation data used to simulate the 3D distributed mixed-mode beams. Results are shown in time and speedup compared to a single GPU.

Table 5.5 shows results for all specimens. Speedup is related to the time on a single GPU. Graph 5.21 shows time results for the reduced-scale (in seconds) and mid-scale specimens (in minutes), by varying the number of compute nodes. According to Graph 5.21, the function is as expected since there is a moment which time does not lower enough for more compute nodes (in this case, five). The function is exponential, as expected. However, for five compute nodes, time is not fast enough if compared to a single GPU. The first mesh is not fine enough to allow a significant speedup. For the finer mesh, this can be due to network bottleneck synchronization by sending and receiving messages between

the communication layer among partitions.

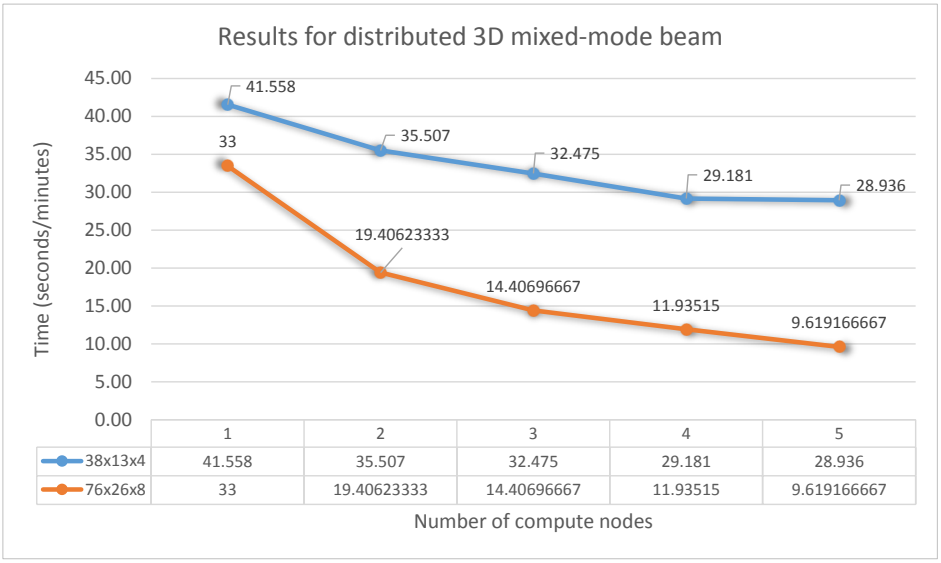

Figure 5.21 – Number of compute nodes versus time Graph for the reduced and mid-scale 3D mixed-mode beam specimens. The time for Beams 38x13x4 and 76x26x8 are in seconds and minutes, respectively.

Figure 5.22 shows the final result for the beam 38x13x4, executed by five compute nodes. The image below shows the beam divided into five partitions and the cohesive elements as red. The fracture propagated at 30°, like in the single GPU case and theoretical model. Because the main crack propagates in that direction, notice that compute node 4 does not contain any cohesive elements and compute node 1 contain few cohesive elements, which cleary indicates an unbalanced kernel distribution among partitions: in this case, the Cohesive Forces kernel and the Insertion of Cohesive Elements kernel.

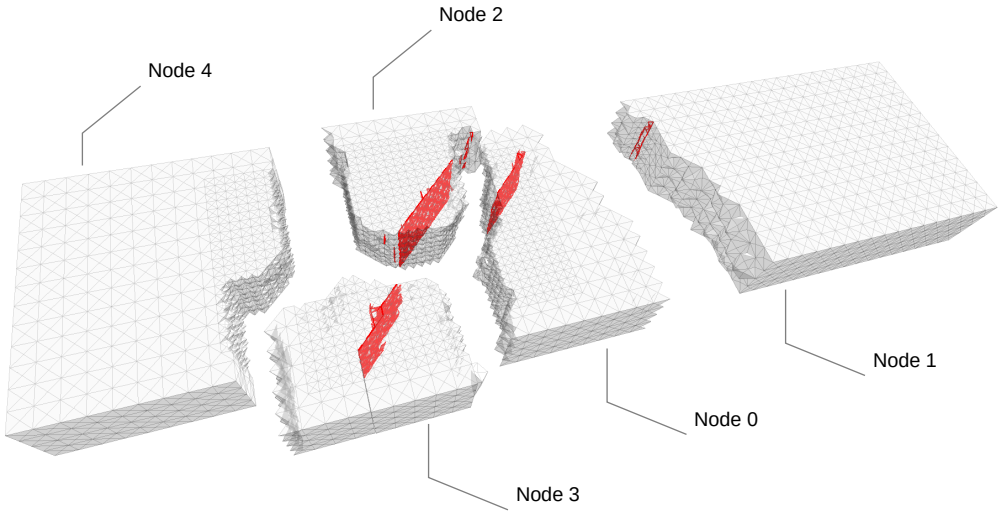

Figure 5.22 – Mesh partitioning at the end of the simulation at t = 270 μs for the reduced-scale 3D mixed-mode beam. Notice the uneven distribution of cohesive elements (shown in red) between partitions.

Graph 5.23 shows the GPU profiling for each node of the reduced-scale $38 \times 13 \times 4$ beam, running in 5 compute nodes, not including message passing from OpenMPI. In the graph, we observe as expected like in Section 5.7: internal and cohesive forces and stress *kernels* dominating the simulation average time. Copying messages back and forth between CPU and GPU occupies an important part of the simulation time, but is not a bottleneck. Internal force kernel times are different between compute nodes because when partitioning the mesh, different number bulk elements and colors can be assigned between different mesh partitions. We can also observe by the cohesive force kernel that compute nodes 1 and 4 practically do not insert cohesive elements in their partition. Therefore, the main branch of the fracture, as expected, is concentrated near the middle-left part of the mesh, near partitions 0, 2, and 3. This also indicates that the Insertion of Cohesive Elements kernel and Cohesive Forces kernel will occupy a greater portion of the simulation time in some nodes, while other nodes continue their tasks and wait for previous compute node tasks are completed.

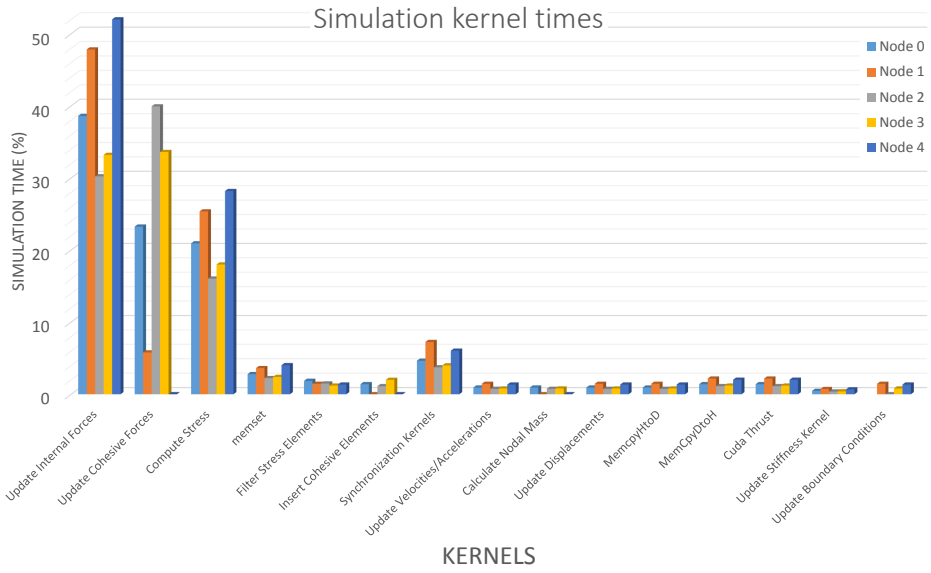

Figure 5.23 – GPU profiling for each node of the reduced-scale 32x13x4 beam, running in 5 compute nodes, not including message passing from OpenMPI.

Graph 5.24 shows for each compute node among 5 running processes for the 3D beam with $76 \times 26 \times 8$ discretization, simulation time occupied by both GPU and CPU and synchronization of nodes. We can observe in this graph the the synchronization stage occupies a great portion of the simulation time. In this stage, we include sending and receiving messages containing nodal stresses, masses, velocities, and accelerations.

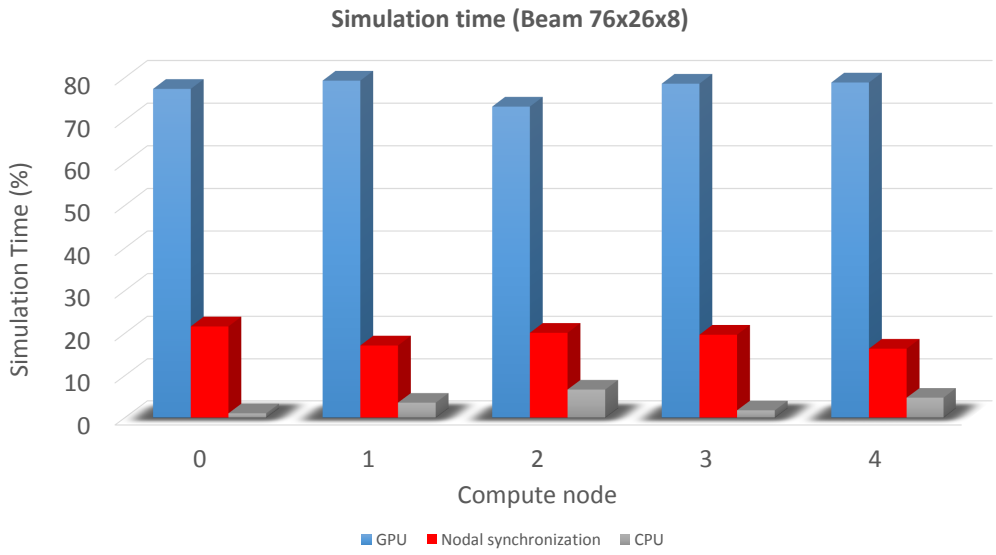

Figure 5.24 – Graph showing the distribution of simulation time in each node between node synchronization and GPU computations for the reduced-scale 3D mixed-mode beam specimen.

Graph 5.25 shows the evolution of kernels along time in CUDA stream. Empty spaces that appear repeatedly in the timeline are the node synchronization and MPI sending and receiving of messages from the host, occuring immediately after the nodes copies the message to the host. Thus we can conclude that one of the bottlenecks of the simulation is the excessive size of the message sent by MPI over network.

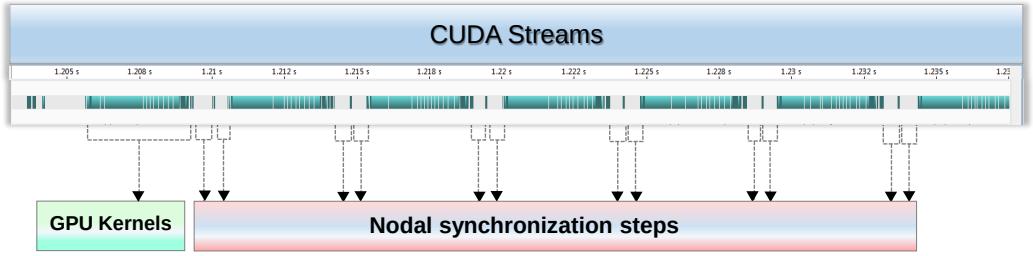

Figure 5.25 – Graph showing the distribution of kernels along time in CUDA streams. Empty spaces that appear repeatedly in the timeline occur immediately after the nodes copy the message to the CPU and immediately before sending them to the receiver.

Figures 5.26 and 5.27 illustrate the results for the simulations of the full-scale specimens at t = 219 μs and t = 220 μs for the coarser and finer meshes, respectively. It is important to notice that the main crack clearly propagates at an angle of about 30° just like the finer versions of the mesh in the single GPU. The first full-scale specimen with approximately 2.3 million bulk elements took 15 minutes to simulate, while the second specimen, with approximately 5.4 million bulk elements, took 82.5 minutes long. Espinha et al [74] simulated a 3D mode-I problem, or a crack separation mode (Opening mode) in which a tensile stress acts normal to the plane of crack. The simulation had 3,840,000 finite elements, 12,000 timesteps and 512 CPU nodes, with a total simulation time of 39.4 minutes. Our simulation deals with mixed-mode problem, which includes the Opening mode with the Sliding mode (shear stress acts parallel to the plane of the crack and perpendicular to the crack front). Considering our simulation time, problem type (mixed-mode), and size, we can say that the increase in performance of the simulation is significant.

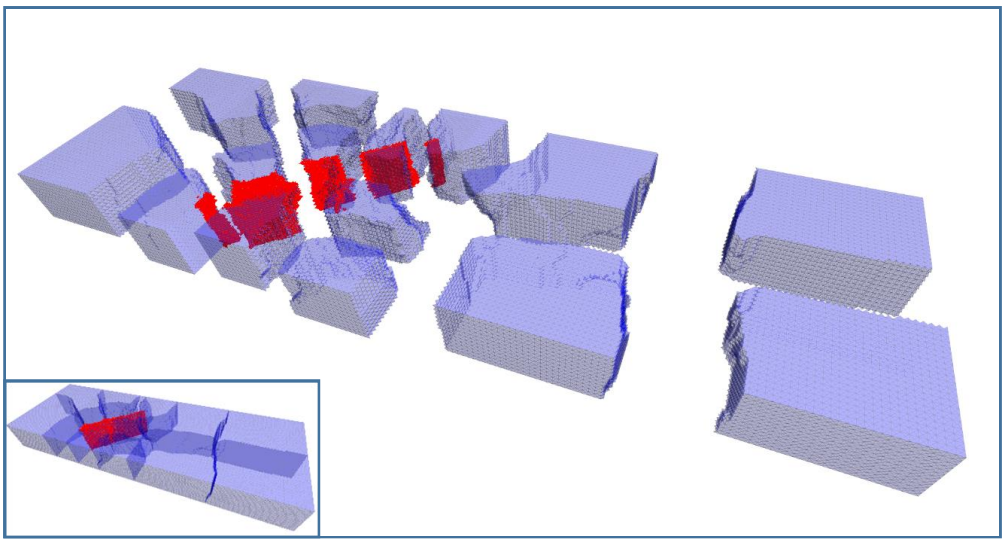

Figure 5.26 – First large-scale 3D mixed-mode beam specimen at simulation end at t = 219 μs with 2,351,424 bulk elements.

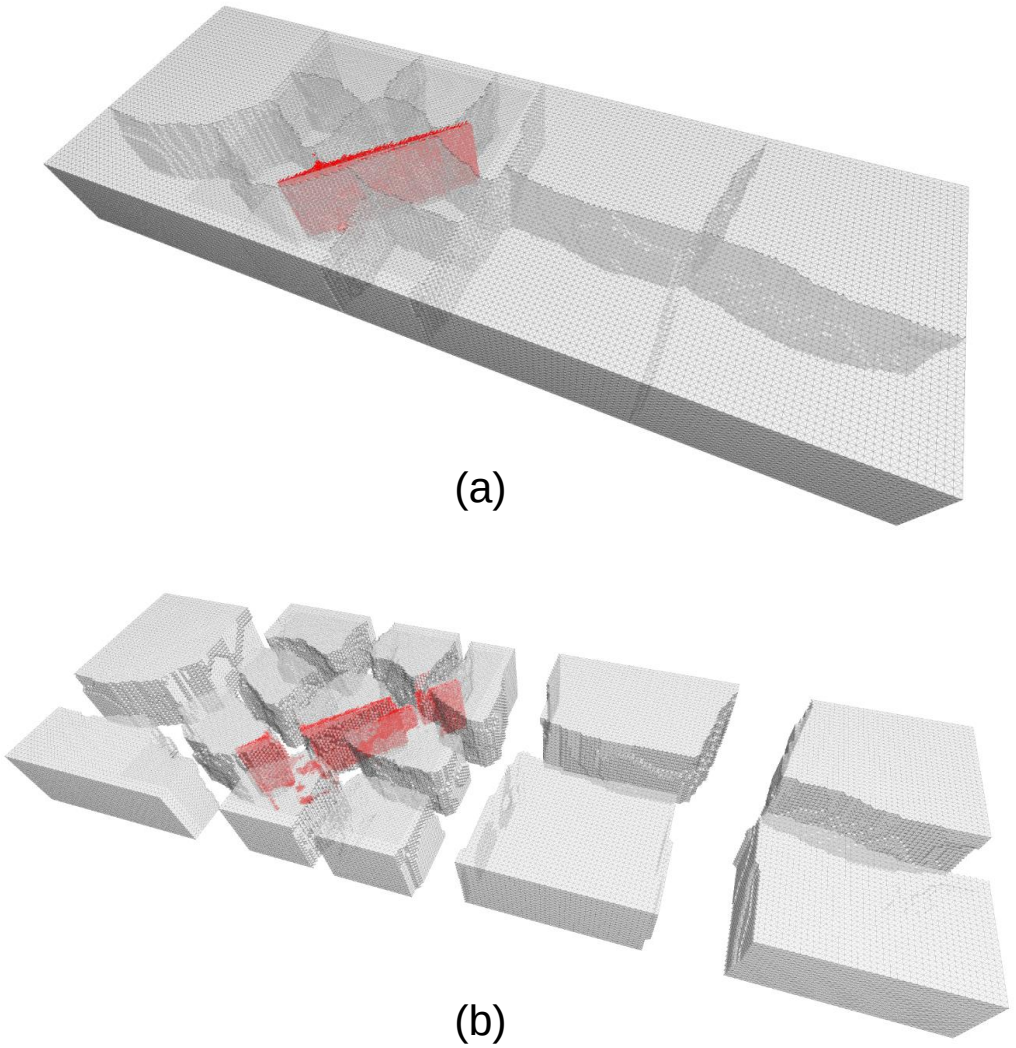

Figure 5.27 – Second large-scale 3D mixed-mode beam specimen at simulation end at t = 220 μs with 5,495,168 bulk elements. Cohesive elements are unevenly distributed, which leads to extra cohesive computations in certain partitions, fewer in some, and idle in others. The main crack propagates like in the results of the single GPU, at 30°.