Generic volume transfer for distributed mesh dynamic repartitioning

Guillaume Damiand, Fabrice Jaillet, Vincent Vidal

Generic Volume Transfer for Distributed Mesh Dynamic Repartitioning

Guillaume Damiand [1*], Fabrice Jaillet [1] and Vincent Vidal [1]

[1*]CNRS, UCBL, INSA Lyon, LIRIS, UMR5205, F-69621 Villeurbanne, France.

*Corresponding author(s). E-mail(s): guillaume.damiand@cnrs.fr;
Contributing authors: fabrice.jaillet@liris.cnrs.fr; vincent.vidal@liris.cnrs.fr;

Abstract

Efficient and distributed adaptive mesh construction and editing pose several challenges, including selecting the appropriate distributed data structure, choosing strategies for distributing computational load, and managing inter-processor communication. Distributed Combinatorial Maps permit the representation and editing of distributed **3**D meshes. This paper addresses computation load and expands communication aspects through volume transfer operation and repartitioning strategies. This work is the first one defining such transfer for cells of any topology. We demonstrate the benefits of our method by presenting a parallel adaptive hexahedral subdivision operation, involving fully generic volumes, in a process including a conversion to conformal mesh and surface fitting. Our experiments compare different strategies using multithreading and MPI implementations to highlight the benefits of volume transfer. Special attention has been paid to generic aspects and adaptability of the framework.

Keywords: Mesh geometry models, Volume mesh, Distributed algorithms, Cell balancing.

1 Introduction

Parallel programming and distributed 3D meshing are of primary importance to tackling a large-scale problem, particularly for numerical simulation. Parallel algorithms based on distributed mesh data structures will seek to optimize available resources by distributing the workload among multiple processors (e.g., multicore processors).

In this field, the most common solution will use an out-of-core partitioning of the mesh. Finding the best global partition is a challenging task, and many strategies have been elaborated to maintain a certain quality in the partition during the mesh modification process. However, the quality criteria are inherently related to the underlying problem to solve, and this process is even questionable [Hen00].

Based on an initial mesh partitioning, a distributed data structure will permit to maintain the global mesh consistency on all the processors during topology modification operations. In addition to supporting inter-processor communication, which may require managing copies of inter-processor boundary elements (*ghost* elements) [HKB21], a distributed global structure should allow efficient access to neighboring cells/blocks (intra and inter-node) [LLM+12] or an owner processor identifier.

Maintaining the distributed mesh information consistency is often not sufficient. For instance, for adaptive mesh refinement, the number of mesh entities associated with a given processor can rapidly grow and unbalance the workload among

available processors. This results in increased computation times as the running time is related to the slowest processor [Hen00]. In this context, dynamic load balancing and cell transfer strategies allow the dynamic assignment of new/old mesh entities to other processors [SS06]. This locally distributes the cells of the global mesh to ensure a better load balance. For block-based adaptive mesh refinement data structure, building a locally computable mapping scheme from blocks to processors avoids performing explicit re-balancing (or rarely) [LLM+12].

Most existing methods require hierarchical data structures or ghost cells, both of which are expensive to maintain and require specific communication strategies. In contrast, we propose a volume transfer operation that eliminates the need for both hierarchical data structures and duplicated ghost cells. Moreover, existing methods deal with some specific type of cells with fixed topology such as [CA15, MPR19], and also [ALSS06, LCW+06] that presented local and global migration experiments similar to ours but restricted to tetrahedral meshes.

In this paper, our contributions consist of the following:

- we define a volume transfer operation with arbitrary element shape and topology, regardless of whether it is conformal or not, capable of transferring from and to any block without any constraint.
- we introduce a dynamic migration procedure based on fast multi-criteria oracles, easily tuned to balance at best loads between computation cores regarding the specificities of each application.
- we rely on a local handling of critical cells (darts), with no *ghost cells*, avoiding duplicated information and subsequent redundant operations.

The rest of this paper is organized as follows. Section 2 presents an overview of related work on distributed mesh data structures, combinatorial maps, and their distributed version. Section 3 deals with the volume transfer operation and associated distributed topological modifications. Section 4 discusses strategies of cell balancing, while Sect. 5 demonstrates the benefits of volume transfers in parallel adaptive mesh refinement experiments.

2 Related Work

There are many parallel data structures for 3D mesh construction and editing. We focus on distributed data structures and implementation that allow for parallel adaptive generation, analysis or editing (e.g., simplification, adaptive refinement). Among the different characteristics of such data structures, we can mention *flexibility* (available volumetric element types, conformality, and manifoldness), *parallel scalability* (when the scale of the problem and the number of cores increase, a scalable algorithm cost will not be dominated by communication), and *compactness* (minimal data structure - *including ghost cells*, ability to compress some data). Generally, a trade-off is a search for scalability while maintaining as much compactness as possible, which equates to a trade-off between time and memory (parallel) complexities.

Adaptive Mesh Refinement (AMR) [BGG+08, JLY10] is a technique that directs computational resources to areas where higher precision is required by refining the mesh in those regions. To locate neighboring blocks and preserve the mesh structure's invariance, conventional parallel AMR implementations retain tree information on each process [LLM+12]. However, the replication of the global mesh structure on each processor can prevent scalability. It is therefore recommended to either store a coarsened overlap (low-res) version on each processor (only "locally owned" domain is fully refined) [BBHK12] or not store any hierarchical global data structure [LLM+12, DGLZD18].

Block-structured AMR [LB10] involves the union of regular grids, allowing the reuse of regular grid sequential codes. In contrast, *unstructured AMR* [FLS+97, LCW+06] provides greater geometric flexibility at the expense of explicitly storing all neighborhood relations between mesh elements. The primary challenge in these mainly tetrahedral-based methods is to maintain element quality as the mesh is coarsened or refined.

Octree is a popular hierarchical representation used for parallel generation or coarsening of hexahedral meshes. However, accessing the neighborhood of a cell is not direct and may be penalizing to produce balanced meshes requiring at most one level of refinement between adjacent cells. Producing conformal hexahedral meshes is also arduous, and mixed elements meshes [JL22, LPC21] or unstructured hybrid meshes [TCL+20] tend to

offer better computational efficiency compared to fully tetrahedral or hexahedral meshes. Additionally, they generally enable better preservation of the mesh surface by employing tetrahedra along the surface border, compared to hexahedra.

Parallel Octrees [BGG+08, SB10] have shown promising scalability and low computational overhead. However, a single octree can encode only a cube-shaped domain. To overcome this limitation, the *Forest-of-Octrees* [BWG11, HBK+23] approach supports the representation of general geometries by assembling multiple octrees. The macro-structure of the forest is shared among all cores. The approach of space-filling curves [JLY10], mostly using Morton, Gray code, Hilbert or Moore ordering for tree traversal, has significantly improved the scalability of parallel octree-based AMR simulations [CDF+03, DHF+09, HS18].

Hierarchical structures can help coarsen the global mesh where needed. The distributed mesh data structure may try to improve cache locality and thus, the performance when accessing corresponding data [NDB17, HKB21]. The distributed mesh data structure can also maintain invariants such that geometrically neighboring cells may differ by only a single refinement level [BBHK12, LLM+12, TCL+20]. That may facilitate the extraction of a conformal mesh but may involve communication with processors that own nearby parts of the mesh.

In a distributed mesh, a unique processor ID is assigned to each mesh element to indicate its current partition within the mesh. The process of cell migration involves reassigning a new processor ID to a selected group of mesh elements and subsequently transferring the corresponding cell data to the memory of the newly designated processor. Cell migration usually achieves load balancing for AMR [LLM+12] in situations where computational requirements vary unpredictably over time. In such cases, workloads can be dynamically allocated to processors to optimize system performance [KK93, HD00, DSS17]. Local heuristics are effective in dealing with localized changes in load balance, as in [LMA15] where splits are performed to minimize interfaces. However, large changes require a global method and a global load balancing strategy.

When considering parallel memory efficiency, there are two key factors to prioritize. Firstly, limiting the amount of data shared between processors is crucial (shared global structure, ghost elements), as this can hinder the scalability of the data structure. Secondly, compressing some shared data and messages [FANF12] can also improve parallel efficiency. These strategies can help to mitigate the impact of communication overhead and improve the overall compactness of parallel computing.

In the majority of the distributed AMR methods, it is worth noticing that the load balancing is performed through cell transfer. Few of them have integrated a dynamic process to exchange cells between computing nodes [ALSS06, SS06, LLM+12], while others are keeping a global representation of the mesh and interface, and where the cells are locally attributed to the corresponding computing node after a global repartitioning step [BGG+08, BGG+10, BBHK12, LCW+06, LMA15, TCL+20].

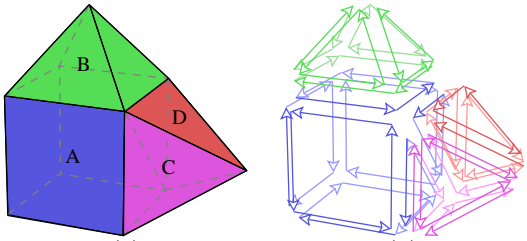
We summarize the limitations of previous work as follows:

- *The use of a global structure*: Storing a global structure on each processor can hinder scalability. Accessing the neighborhood of a cell may require additional memory and can be penalizing to produce balanced meshes that require at most one level of refinement between adjacent cells.
- *Restricted element type*: Maintaining element quality in pure tetrahedra methods as the mesh is coarsened or refined is complex. Pure hexahedra methods offer better computational efficiency, but producing conformality is arduous [LPC21], and hexahedra do not allow satisfactory surface preservation [PCS+22].
- *Ghost elements*: The use of ghost elements can impact the performance and complicate the load balancing.

In this paper, we solve these limitations: (1) we avoid the use of a global structure; (2) we do not limit the element type; (3) we do not use ghost element while offering load balancing.

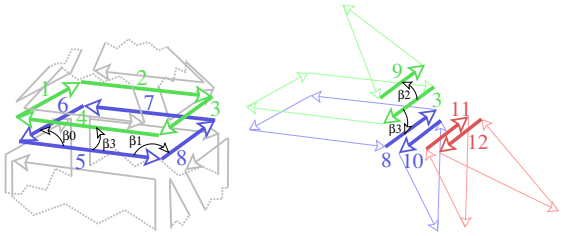
2.1 Combinatorial Maps

A 3D combinatorial map [Lie94, DL14], called 3-map, is a combinatorial data structure that allows representing a 3D mesh decomposed into cells: vertices, edges, faces, and volumes; plus adjacency



(a) (b)

Fig. 1: Example of: (a) a 3D object; (b) the corresponding 3-map.



(a) (b)

Fig. 2: Zoom on: (a) the face between volumes `A` and `B` of Fig. 1b; (b) edge incident to volumes `A`, `B`, and `D` of Fig. 1b.

and incidence relation between all the cells. Each edge of the mesh is decomposed into basic unit elements: the *darts*. For each edge, there is one dart per face and per volume incident to the edge. The connectivity between the cells is encoded through mappings between darts. For each dart d, $\beta_1(d)$ gives the next dart following d in the same face and the same volume; $\beta_2(d)$ gives the other dart of the same edge, the same volume but the adjacent face and $\beta_3(d)$ gives the other dart of the same edge, the same face but the adjacent volume. β_0 is the reverse relation of β_1, allowing to turn around darts of a face in the reverse direction. Note that β_2 and β_3 are involutions, i.e. one to one mappings equal to their inverse. All the cells and all the adjacency and incidence relations in a 3-map can be retrieved using the darts and their links.

A 3-map can be seen as a generalization in 3D of the well known half-edge data structure [Wei85]. Half-edges correspond to darts, next to β_1, previous to β_0 and opposite to β_2. β_3 can be seen as a second opposite relation but between volumes instead of faces.

Let us have a look at the 3D mesh shown in Fig. 1a. This 3D mesh is composed of 4 volumes: 2 tetrahedra (labeled `C` and `D`), one pyramid (labeled `B`) and one hexahedron-like element (labeled `A`), which is not really a hexahedron because one of its faces is split in two triangles. 3-map are naturally able to represent any kind of polyhedra. And, for example, `A` is adjacent to `B`, `C`, and `D`; and `C` is adjacent to `A` and `D` (note that adjacency is a symmetric relation).

This mesh is described by the 3-map shown in Fig. 1b, where darts are drawn by oriented segments. `A` is represented by 26 darts, `B` by 16 darts, and `C` and `D` by 12 darts each.

Let us zoom in on the face between volumes `A` and `B` (Fig. 2a) to observe more precisely the darts and their links. This face has 8 darts, numbered from 1 to 8. And for example, $\beta_0(5) = 6$, $\beta_1(5) = 8$ and $\beta_3(5) = 4$. If we zoom in on the edge between volumes `A`, `B`, and `D` (Fig. 2b), we can observe that it is represented by 6 darts. We have for example $\beta_2(3) = 9$ and $\beta_3(3) = 8$.

A 3-map can represent objects with or without boundaries. In Fig. 1, some faces belong to the border of the mesh: they are incident to only one volume. Darts representing these faces are said 3-free (for example, dart 12 in Fig. 2b). Non-3-free darts are said 3-sewn, and we say that dart d is i-sewn with dart d' when $\beta_i(d) = d'$.

One main interest of 3-maps is to fully describe the mesh topology while allowing for mixing of different face and volume types (which are not restricted to some specific topology like triangles, quads, tetrahedra, or hexahedra). Another main interest is that there are many operations to build and modify a 3-map. These operations are often defined locally and thus have good computational time complexity.

Note that the definition of combinatorial maps is more generic since they are defined in any dimension, and they allow borders for each dimension. Interested readers can refer to [DL14] for more explanations and definitions.

2.2 Distributed Combinatorial Maps

In [DGLZD18], a distributed version of 3-maps is defined. Its main principle is to split a combinatorial map into several independent parts (called *blocks*), such that all darts that belong to the same volume (cell) must belong to the same block. Ids are associated with darts and faces that belong

to the boundary of a block to retrieve the corresponding darts and faces in the other block. These blocks define a partition of the global 3D mesh.

More precisely, if a dart d belongs to a different block than the one of $d' = \beta_3(d)$, d and d' are called *critical darts*. Ids are associated with these critical darts (denoted $cid(d)$), such that each pair of darts d and $d' = \beta_3(d)$ have the same label. Reciprocally, if two critical darts d and d' share the same id, then we must have $d' = \beta_3(d)$. In addition, a face is called critical if its darts are critical. Ids are also associated with critical faces (denoted also $cid(f)$). For two critical darts such that $d' = \beta_3(d)$, then the ids of the two opposite critical faces containing them are the same.

Each block stores its own set of critical darts and critical faces. These two sets allow: (1) given a critical cell id, to retrieve one of its darts (denoted $dart(id)$), for example to serve as starting point for an operation; (2) given a critical cell id, to retrieve the other block containing this cell too, denoted $other\_block(id)$, for example to send it a message.

It is crucial to keep in mind that any label can be assigned to critical cells, but two critical cells must share the same label if and only if they correspond to the same cell in the two blocks.

Let us illustrate the notions around distributed 3-maps thanks to the example of Fig. 3, which shows one distributed version of the 3-map of the previous example (in Fig. 1). This example has 3 blocks called B_0, B_1, and B_2, drawn in different colors. 3 faces are critical, labeled [a], [b], and [c], and 18 darts are critical. For example, [a] between blocks B_0 and B_1, has 3 critical darts in each block, labeled 1, 2, and 3. In examples and figures, we use [id] to denote critical face ids in order to distinguish them from critical darts ids. In block B_0, we have for example $other\_block([a]) = other\_block(1) = B_1$ (and reciprocaly in block B_1, $other\_block([a]) = other\_block(1) = B_0$).

In [DGLZD18], a distributed 3-map is used to propose a parallel adaptive hexahedral mesh refinement algorithm. Each block subdivides its part independently, and messages are exchanged when critical darts and critical faces are subdivided to ensure the global consistency of the distributed 3-map. One potential limitation of this work is that the number of volumes in each block may vary depending on the mesh, leading to significant load imbalances among processes.

In this paper, we solve this limitation by proposing two improvements: (1) we define a new operation allowing the transfer of a volume between two different blocks; (2) we use this transfer operation to balance the loads among the different cores while performing parallel operations for 3D mixed elements mesh editing. Note that, to our knowledge, this is the first method allowing the transfer of cells with any topology. To demonstrate its potential, we illustrate the cell transfer in an adaptive hexahedral mesh refinement algorithm, including converting non-conformal cells into transitional patterns of mixed elements (potentially composed by tetrahedra, hexahedra, prisms, and pyramids) and finally projecting external points on the surface.

3 Volume Cell Transfer

Let us consider a distributed 3-map having n different blocks $B_0, \ldots, B_{n-1}$. The goal of the volume transfer operation is to send one volume v from block B_i to block B_j (with $i \neq j$) while preserving the partition property. We define this operation to be generic, without any constraint on block B_j's location, the topology of the volume, or its adjacent volumes and critical faces.

In the following, we suppose that each block is associated to a unique process, and that the only possible communication will be by sending messages. This framework is a little bit too restrictive for multi-threading architectures, which allow shared memory, but this enables us to develop the same software for both multi-thread and multi-processes architectures, changing only the communication part, as we will see in the experiments section.

Given a volume v in block B_i, each face of v belongs exactly to one of the sets that we denote by:

- ∂_v the non-critical faces of v that are 3-free (these faces belong to the boundary of the mesh);
- F_v the non-critical faces of v that are not 3-free (these faces are internal to block B_i);
- F_{vj} the critical faces of v between blocks B_i and B_j;
- F_{vO} the critical faces of v between blocks B_i and B_k, $k \neq i$ and $k \neq j$ (O stands for others).

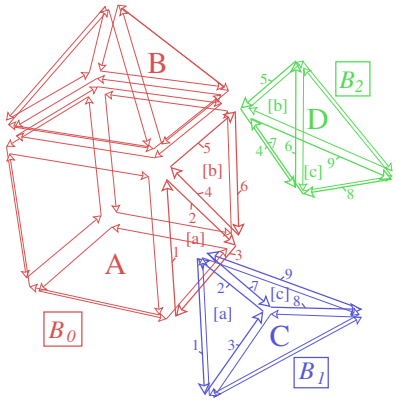


Fig. 3: Example of a distributed 3-map representing the 3-map shown in Fig. 1b.

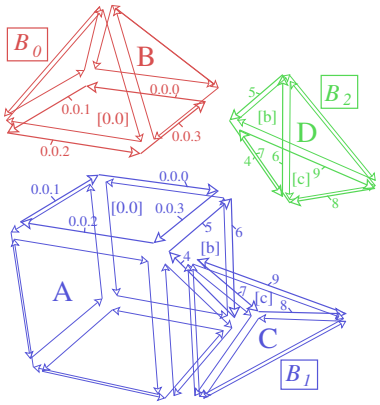


Fig. 4: Result of the transfer of volume A from block B_0 from the distributed 3-map of Fig. 3 into block B_1.

Migrating volume v from block B_i to block B_j requires the following modifications, detailed in the following:

1. **move the volume**: remove v from B_i (geometry and topology), and add it into B_j;
2. **attach the volume**: 3-sew all critical faces of F_{vj} with their corresponding critical face in B_j; remove the critical status of these faces (which become internal to block B_j);
3. **create new critical faces**: mark all internal faces of F_v as critical in blocks B_i and B_j, using new critical ids;
4. **update remaining critical faces**: update the critical faces with other blocks in F_{vO}: for all concerned blocks B_k, update the opposite block of these faces, which becomes B_j; and remove these critical faces from the local list in B_i and add them in B_j (keeping existing ids).

Let us detail this process on an example, considering volume A in Fig. 3 that belongs to block B_0 and that we want to transfer into block B_1. This volume has 7 faces, 4 of which are 3-free and thus belong to ∂_A. The face (square) between A and B is internal to block B_0 and thus belongs to F_A. The critical face [a] is between blocks B_0 and B_1 and thus belongs to F_{A1}, while the critical face [b] between blocks B_0 and B_2 belongs to F_{AO}. Figure 4 shows the result of the transfer. We can observe that: (1) the face [a] $\in F_{A1}$ is no more critical. `A` and `C` are now 3-sewn along this face; (2) [b] $\in F_{AO}$, its other block is changed in B_2 : $other\_block([b]) \leftarrow B_1$ (it was B_0 before the transfer); (3) a new critical face [0.0] is created (associated with the face in F_A) between blocks B_0 and B_1.

Note that [0.0] is a new label generated by the transfer, while [a], [b], and [c] are existing labels (from the combinatorial map in input) that can be reused as is by our method. By construction, those two families of labels are distinct (see Sect. 3.1).

So, in each block, we have the following critical faces before transferring volume `A`.

- in block B_0: {[a], [b]};
- in block B_1: {[a], [c]};
- in block B_2: {[b], [c]}.

and after transfer of `A` to Block B_1:

- in block B_0: {[0.0]};
- in block B_1: {[0.0], [b], [c]};
- in block B_2: {[b], [c]}.

In Fig. 5, you can see how blocks communicate during this operation. There are two types of messages. B_i sends a long message to B_j, of type `TR` (for "TRansfer"), containing 4 parts: (1) the geometry and the topology of the cell; (2) the current list of critical faces between blocks B_i and B_j that will serve to attach the volume with its neighbors after the transfer; (3) the list of new critical faces between B_i and B_j; (4) the current list of other critical darts and faces, and the id of their other block.

A second type of message (type `UP` for "UPdate other block") is sent from B_i to other blocks B_k, $k \neq i$ and $k \neq j$, that share one critical face with

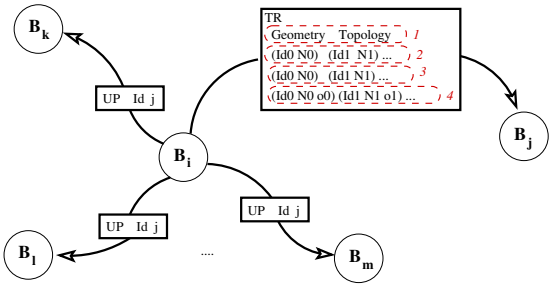


Fig. 5: Summary of the different messages exchanged for the volume transfer operation. There are two types of messages: `TR` for "TRansfer" and `UP` for "UPdate other block".

v. This short message contains only the critical id of the face and the id of the block where v is transferred (j).

We can observe that messages of type `UP` will always remain short. However, the message of type `TR` can be quite large due to the inclusion of geometry, topology, and potentially several critical ids. For example, if one of the initial faces in a cell has been highly subdivided, but not in its opposite cell, this is leading to many critical darts and faces. That is the price to pay for dealing with cells of any topology, as there is no means to retrieve this information from an unstructured representation. But in most cases, this message remains of reasonable size, especially for 1-balanced conformal meshes that tend to lower this problem.

In the following sections, we detail the different operations achieved by the distinctive blocks: source B_i, destination B_j, and relatives B_k if any, with $k \neq i$ and $k \neq j$. Finally, the whole process is illustrated on an example in Sect. 3.4.

3.1 Operations in source Block B_i

The main principle of the transfer operation for block B_i is the following: (1) start to build a message containing all the information required to re-create v in B_j; (2) update its critical cells; (3) remove v from this block mesh. This process is summarized in Algorithm 3.1. The `TR` message built by B_i contains 4 different parts detailed in the 4 following subsections. The `UP` message appears in the last of these 4.

Algorithm 3.1: Transfer operations in block B_i.

Result: Updating Block B_i;
Send messages `TR` to Block j and `UP` to Blocks k.

`// message TR part 1`
Encode geometry and topology of v in `TR`;
`// message TR part 2`
foreach *critical face* f *in* F_{vj} **do**
  Encode $cid(f)$ and the local index of one of its dart in `TR`;
  Unset critical face f and all its critical darts;
`// message TR part 3`
foreach *internal face* f *in* F_v **do**
  $cid(f) \leftarrow$ [i.#new] ;
  $other\_block(f) \leftarrow B_j$;
  Encode [i.#new] and the local index of one starting dart of f in `TR` ;
  Label all darts of f with $i.\#new.q$, starting from $q = 0$, in direct order (using β_1);
  ++ #new ;
`// message TR part 4, and message UP`
foreach *critical face* f *in* F_{vO} **do**
  Encode $cid(f)$ and j in `UP`;
  Send `UP` message to Block $other\_block(f)$;
  Encode $cid(f)$, $other\_block(f)$ and the local index of one starting dart n in `TR` ;
  Encode the critical id of all darts of f, starting from n, in direct order in `TR` ;
  Unset critical face f and all its critical darts ;
Send `TR` message to Block B_j ;
Remove v from B_i ;

3.1.1 Geometry and topology, `TR` part 1

The geometry and the topology of v come first. The encoding is local and independent of the rest of the mesh and is generic enough to be able to describe any geometry and topology.

First, all the vertices of v are numbered locally to the element (starting from 0). Next, each face of v can be described by the ordered list of its vertices (using their indices). Note that the order of vertices, the order of faces, and the starting vertex for each face can be chosen arbitrarily.

This encoding of v gives an implicit numbering of the darts. Taking the first face, the dart associated with the first pair of vertices is numbered 0, the dart associated with the second pair of vertices is numbered 1, and so on.

The topology of the faces is explicitly stored in this encoding through the ordered list of vertices, while the topology of the volume is implicitly represented: two faces are adjacent when they share the same pair of indices in reverse order.

3.1.2 Critical faces between blocks B_i and B_j, TR part 2

The critical faces in F_{vj} will be no more critical after the transfer of v, they become internal to B_j according to the process exposed below.

The new volume v in block B_j must be attached to its neighbors in B_j. Before the transfer, v was virtually attached to these neighbors through its critical faces between blocks B_i and B_j that belong to F_{vj}. The goal here is, for each face in F_{vj}, to find one dart of v in this face and the corresponding dart of another volume of B_j that must be 3-sewn.

This is achieved thanks to the id associated with critical darts. One pair (cid, n) is added at the end of the message for each critical face in F_{vj}, with cid being the critical id of one dart of the face and n being the local index of this dart computed in the previous step.

3.1.3 New critical faces between blocks B_i and B_j, TR part 3

After the transfer of v from B_i to B_j, the faces of v in B_i that were internal to this block (i.e., 3-sewn with another volume of B_i) will become critical between blocks B_i and B_j. New critical faces must be created while ensuring consistent labeling in both blocks. This consistency is crucial to ensure the global validity of the mesh through its different blocks.

These new critical faces belong to F_v. For these faces, we need to create new ids and ensure their uniqueness. To do so, each block stores $\#new$, its proper number of new ids already defined and used during previous steps of volume transfers. A new critical face is labeled [i.#new], i being the block number. With this labeling, each new critical id is globally unique, since $\#new$ is incremented after each creation of a new critical face. Moreover, the process of creating a new id is local to each block without the need for any global communication between blocks, which is a key point in our method.

These critical ids are created in block B_i and added at the end of the message (using like for the previous case one pair (cid, n) for each new critical face, with cid the new critical id, and n the local index of one dart of the face).

B_i labels its new critical faces using these ids. Critical darts of these faces must also be labeled consistently in both blocks. To do so, B_i labels each new critical dart by i.#new.q, with [i.#new] the id of the face and q being the position of the dart in this face starting from the dart used in the pair (cid, n), and using β_1 to traverse all the darts of the face. This labeling avoids to add the labels of these critical darts in the message since they can be retrieved directly using the label of the critical face.

3.1.4 Critical faces between blocks B_i and B_k, TR part 4 and UP

The last required modification is that of the old critical faces between v and other blocks (different from B_i and B_j). For each face in F_{vO}, we:

1. send a message to block B_k containing the critical face id and j, the id of the block where v will be transferred;
2. remove these critical faces and their critical darts from B_i, which are no longer critical in this block;
3. add at the end of the message to block B_j the critical ids of these faces and the critical ids of their darts, with their other block, and the local index of their darts. Unlike the case of new critical faces (explained in the previous section), we do not have to create new ids, we can reuse the existing ids since they already exist in blocks B_k. Also, note that we need to send all the critical ids here, as we cannot assume any specific property on the critical ids of darts or faces.

3.1.5 Update of B_i

Once the message TR is built, it is sent to B_j. The critical faces of v and their critical darts are removed from the associative arrays in B_i since they will be no more critical after the transfer of

v. Then, B_i removes v from its mesh using the removal operation for combinatorial maps [DL14]. This operation removes all the darts of v and updates the β_3 relations of darts that were linked with v.

3.2 Operations in destination Block B_j

When B_j receives the message from B_i, it can recreate v from its encoding and add it to its mesh partition. During the reconstruction, the ids of each new dart are stored (according to the order of faces and their list of vertices, as explained above). For now, v is isolated, namely disconnected from the other parts of the mesh in block B_j. Information contained in the message will be used to attach v with its neighbors and to create new critical elements (Algorithm 3.2).

Firstly, for each critical face in F_{vj}, we retrieve two darts: (1) d_1, the new dart of v using n and the local index of darts of v; (2) d_2, the corresponding dart in B_j, directly using the id of the critical dart and the associative array in B_j that gives for each critical id one of its darts. It is then enough to call the 3-sew operation starting from these two darts, which will correctly link two by two all the darts of their faces ([DL14] for more details on this operation).

After considering all critical faces in F_{vj}, v is now correctly attached to all of its neighbors. These critical faces are no longer critical since v now belongs to the same block B_j as these neighbors. Consequently, all critical ids of these faces and their darts are removed from the associative arrays.

Secondly, for each new critical face in F_v, B_j labels the face using the pair (cid, n) that gives the id of the face and one dart. Critical darts are labeled using n as starting dart and cid as prefix. Starting from the dart n, all the darts of the face are labeled by cid.q, with q the position of the dart in this face, but using here the reverse relation β_0 to ensure that two corresponding darts in B_i and B_j have the same label.

Thirdly, new critical darts and critical faces are created for each critical face in F_{vO}. Here, we use the critical id of the element, the local index of one of its darts, and the id of their other blocks that are directly read in the message.

Algorithm 3.2: Transfer operations in block B_j.

Result: On reception of a `TR` message from Block B_i, updating Block B_j.

1 Create volume v using topology and geometry information (`TR` part 1);
`// 3-sew v to its adjacent faces`
2 **foreach** *critical dart cid of local index n in `TR` part 2* **do**
3 $m \leftarrow$ getDartIndexForLabel(*cid*) ;
4 3-sew(n, m);
5 Unset critical face containing dart m and all its critical darts;
`// Label new critical faces and darts`
6 **foreach** *critical face [cid] with local index n of starting dart in `TR` part 3* **do**
7 Set face containg dart n as critical ;
8 *other_block*([cid]) $\leftarrow B_i$;
9 Label all darts of [cid] with *cid.q*, starting from dart n, $q = 0$, in reverse order (using β_0);
`// Update other critical faces and darts`
10 **foreach** *critical face [cid] with local index n of starting dart in `TR` part 4* **do**
11 *other_block*([cid]) $\leftarrow B_k$ (and the same for all the darts of the face);

3.3 Operations in relatives Blocks B_k with $k \neq i$ and $k \neq j$

The only modification required for all the blocks B_k is to change the other block id from B_i to B_j for the critical faces received in the message, and for all of its critical darts (Algorithm 3.3).

Algorithm 3.3: Transfer operations in block B_k.

Result: On reception of a `UP` message from Block B_i, updating Block B_k.

`// Update critical faces and darts`
1 *other_block*([cid]) $\leftarrow B_j$ (and the same for all the darts of the face) ;

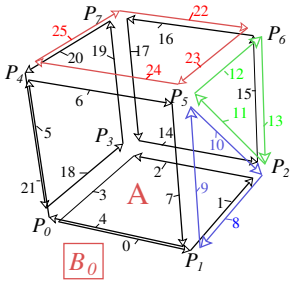


Fig. 6: Possible local encoding of points and darts for Volume `A` of Fig. 3.

3.4 Example

Let us reconsider the example given in Fig. 3 (before transfer) and Fig. 4 (after transfer). One possible encoding of the volume A, which has 8 vertices and 7 faces, is the following (Fig. 6), and this constitutes part 1 of the `TR` message:

```
8 7 # Number of vertices / faces
# Coordinates of each vertex Pi
0 0 0     1 0 0     1 1 0     0 1 0
0 0 1     1 0 1     1 1 1     0 1 1
# Number of vertices and indices of each face
4 0 1 2 3      4 1 0 4 5      3 2 1 5      3 2 5 6
4 3 2 6 7      4 0 3 7 4      4 7 6 5 4
```

In this encoding, critical dart 1 of Fig. 3 is, for example, locally numbered 9 for his element (because it is associated with the second edge of the third face, see Fig. 6 and Sect. 3.1).

Only one critical face belongs to F_{A1}: [a]. The 3 critical darts with ids 1, 2, and 3 correspond to local indexes 9, 10, and 8 of the third face, respectively. It is sufficient to add one of these darts at the end of the message: for example, the critical dart with id 1 and local index 9: part 2 of the message is thus (1 9). When B_1 receives the message, and after having recreated `A` in its block, it can retrieve the two face-opposite darts corresponding to the critical id 1 and 3-sew the two corresponding faces.

One new critical face is in F_A and created (the square between volumes `A` and `B`). This face is labeled [0.0] in blocks B_0 and B_1, following the naming convention [i.#new] with $\#new = 0$, the first critical face to be created in block $i = 0$. One dart of this face is chosen and labeled 0.0.0 in B_0, and the next darts of the face are labeled respectively 0.0.1, 0.0.2, and 0.0.3. In the message, only the number of the starting dart and the critical id of the face are sent, other information will be retrieved. The labeling of new critical darts in B_1 is done starting from the initial dart and using β_0. Part 3 of the message is thus ([0.0] 22).

It remains one critical face to treat, which belongs to F_{AO}, and labeled [b] in the example: (1) in B_2, *other_block*([b]) $\leftarrow B_1$, and the same for all of its critical darts 4, 5, 6 (it was B_0 before the transfer). This is achieved by sending the `UP` message from B_0 to B_2: ([b] 1); (2) these critical faces and darts are removed from B_0; (3) these critical darts and faces are created in B_1 with the same ids and the same other blocks (present in the message, here 2). These critical elements are associated with new darts in B_1, identified by their critical id (here 4, 5, 6), starting with local dart of index 11. Part 4 of the message is thus ([b] 2 11 4 5 6), and that ends the `TR` message.

3.5 Discussion about concurrent accesses

The different blocks are fully independent, they can be traversed in parallel with no concurrent access issues. For volume transfer, it is possible to:

- simultaneously transfer one volume from B_i to B_j and one volume from B_k to B_l if the 4 blocks are different (i.e., $i \neq k$, $i \neq l$, $j \neq k$, and $j \neq l$) and if the two transferred volumes do not share a common critical face (i.e. they are not adjacent in the virtual entire mesh);
- simultaneously transfer one volume from B_i to B_j and one volume from B_i to B_l if $j \neq l$ and if the two volumes are not adjacent in B_i;
- simultaneously transfer one volume from B_i to B_j and one volume from B_k to B_j if $i \neq k$ and if the two volumes will not become adjacent in B_j;
- simultaneously transfer one volume from B_i to B_j and one volume from B_i and B_j if the two volumes are not adjacent in B_i and will not become adjacent in B_j.

It is not yet possible to transfer simultaneously two adjacent volumes from or into the same block since in this case the critical faces may be updated incorrectly.

4 Cell Balancing in Distributed Mesh

The volume transfer we just described can serve as an elementary operation in various mesh repartitioning applications. For example, it is possible to distribute cells among the different blocks in order to balance the workload among all the processes according to some criteria. To this end, we define a generic method for cell balancing, summarized in Algo. 4.1. This algorithm is generic, it takes as input an oracle that tells, for each cell, if it must be balanced. For instance, this oracle can always return true if all mesh cells need balancing, or return true or false according to a specific criterion, like whether the cell intersect the surface or lies in a Region of Interest (see next section for detailed examples). Note that the oracle must depend only on the given volume (and not on any other ones, like neighbors) in order to guarantee a correct incremental updating of the number of volumes to balance.

Algorithm 4.1: Balance cells in block B_i.

Data: An oracle O.
Result: Block B_i contains the mean of cells satisfying O.

```
// Ask all blocks their number of
   volume cells satisfying O
nbVolumesToBalance ← getNbVolumesInAllBlocks(O);
meanNbCells ← mean of nbVolumesToBalance;
for k ← 0 to number of blocks-1 do
    if k = i then
        while nbVolumeToBalance[i] > meanNbCells do
            n ← id of one block in deficit of volumes;
            v ← oneVolumeSatisfying(O);
            transferOneVolume(v, n);
            − − nbVolumeToBalance[i];
            + + nbVolumeToBalance[n];
    Barrier();
```

The first step of this algorithm consists in computing the number of cells that need balancing in each block. For this purpose, each block computes its own number, and these numbers are shared among the different blocks through message passing. After this step, block B_i knows if it has too many cells by comparing its number of cells with the mean of the numbers of all blocks. When this occurs, B_i transfers some of its volumes to other blocks that are in deficit. As our volume transfer works for any configuration, there is no need to transfer a volume to an adjacent block, and thus this transfer can be done directly into any block in deficit. After the transfer, the number of volumes to balance in the two blocks involved in the transfer are updated locally, and the process restarts until block B_i no longer has a volume excess.

To prevent the problem of concurrent access (explained in Sect. 3.5), the transfer is done successively by each block. It avoids the case where two blocks want to transfer two volumes to the same block, while the case where two volumes of the same block are to be transferred at the same time is not possible since there is only one process associated with each block. This is done thanks to the first loop of Algo. 4.1. One block at a time performs the balancing, the other ones waiting at the barrier.

When all blocks have finished this process, the number of volumes satisfying the criteria in each block is balanced (plus or minus one). One main advantage of Algo. 4.1 is to be fully local for each block (except at first step that collects all the number of cells, but this is fast since only few numbers are exchanged). Contrarily, it is more tedious to guarantee specific conditions on mesh topology in each block, as it will be seen in our experiments.

Numerous criteria or oracles on volumes can be employed according to the requirements of each application. Balancing all volumes is one possibility, while another may involve balancing volumes intersected by a surface, particularly in scenarios where a hexahedral mesh is generated to approximate a specified surface. Utilizing such criteria optimizes the workload distribution among different processes, primarily attributed to the subdivision of volumes intersecting the surface. We can also balance the cell volume of the meshes in the different blocks. In this case, the collected numbers are the volumes of the elements and not their numbers, and the criteria to transfer a volume must take into account the size of the volume to transfer.

Various strategies can be defined for transferring volumes, including considerations about geometry (shape or distance) or topology (connected components or holes in final partition). In this paper, we will use the two strategies that follow.

Baseline:

The baseline implementation transfers volumes in excess to the first other block experiencing a volume deficit without any specific criterion or order (called strategy S_1).

Minimizing the number of Critical Cells:

In this version, a priority queue is utilized based on the number of critical faces shared within the same block (referred to as strategy S_2). Specifically, the prioritization involves transferring volumes to neighboring blocks experiencing a volume deficit, with priority given to those that share the most critical faces. Following this, the remaining volumes sharing possible critical faces with any other block facing a volume deficit are subsequently transferred. The priority queue is updated after each transfer to account for modifications in neighborhood relations.

This method prioritizes the transfer of volumes that share critical faces, thereby preventing an increase in the number of holes or connected components within each block. When a volume v in B_i is transferred in a block B_j sharing at least one critical face, v will be attached at least through this face in its new block, thus avoiding the creation of a new connected component. Additionally, a volume is only transferred if it has one critical face in B_i, which is 3-free in B_i (meaning this face has no volume adjacent to it in B_i). Consequently, removing v from B_i will not create an hole, but will only modify the boundary surface.

To ensure a balanced partition upon completion of the process, an optional final step could be set to enable transfer between any block without considering existing adjacency, utilizing the baseline implementation.

5 Experimental Results

In this section, we describe our data set and illustrate the interest of the cell transfer operation and of the volume cell balancing method in an application that generates a conformal 3D volume mesh from a surface. It is important to advise that this serves only as an illustration. In this paper, our goal is neither to introduce a new generation algorithm nor to define the best method for doing so. Our experiments have been developed to evaluate the impact of the cell balancing on computation times and number of cells in the final mesh, not to study the quality of the produced meshes.

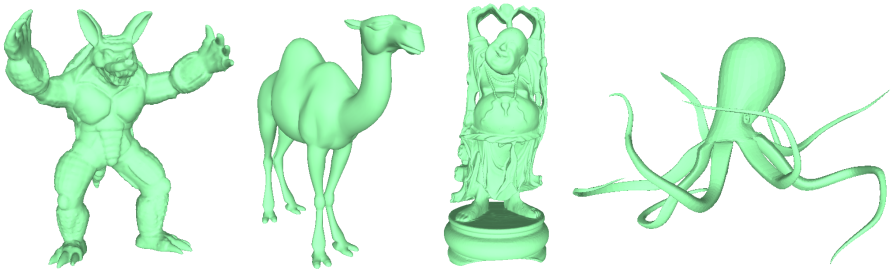
5.1 Data set and Algorithms

We implemented a 3D conformal mesh generation that approximates a given surface, using a distributed 3-map based on the CGAL implementation of combinatorial maps [Dam11]. Our software has been developed in C++, with two different versions: multi-threaded and multi-processing based on MPI (Message Passing Interface). The two versions share the same code except for the communication part.

Our experiment includes the 3 following successive steps:

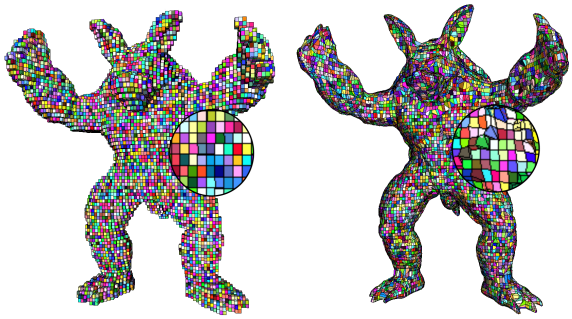
1. create an adaptive hexahedral mesh (using [DGLZD18]);
2. turn the mesh into a conformal mesh by locally subdividing each volume which is not a pure hexahedron, *ie.* with subdivided faces (using [DN22]);
3. project on the surface all the outside points. Remove degenerated or inverted volumes created by the projection.

Details of the different algorithms are given below. The result of the 3 steps of the 3D mesh generation is a 3D conformal mesh, mixing hexahedra and possibly tetrahedra, prisms, and pyramids, that approximates the volume defined by the input surface. So, we can use our volume cell transfer operation presented in Algorithm 4.1 to balance cells between blocks during the subdivision loop of the hexahedra. Note that in the experiment, the cell balancing is activated only for the first step of the mesh generation method (generation of the adaptive hexahedral mesh). This step creates many new cells and consequently will greatly benefit from the cell balancing effects. The other steps (making the mesh conformal and projection of external points on the surface) will not create



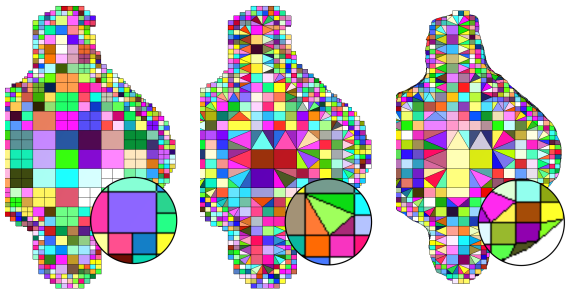
(a) *Armadillo* (b) *Camel* (c) *Happy* (d) *Octopus*

Fig. 7: Data set models. For meshes (a-d), the number of triangles are, respectively, 52,000, 19,536, 1,087,716, and 33,104; and their number of vertices are 26,002, 9,770, 543,652, and 16,554.



(a) steps 1+2 (b) step 3

Fig. 8: The *Armadillo* surface mesh after (a) step 1 and 2; (b) step 3 (with one random color associated with each volume). Note that the meshes after step 1 and step 2 share the same external view, but with different interior volumes (Fig. 9 for a cross-section view).



(a) (b) (c)

Fig. 9: Zoom on the results obtained for armadillo surface mesh, showing the interior of the mesh after (a) step 1, we can see hexahedra with different levels of subdivisions, leading to some T-junctions; (b) step 2, we can check that the mesh is conformal; (c) step 3, only the boundary is modified.

many cells and not change enough the cell balancing between the different blocks to require another balance step. See Sect. 6 for a complementary discussion on this point.

We applied the 3 steps on the 4 meshes shown in Fig. 7, namely *Armadillo*, *Camel*, *Happy* and *Octopus*. Fig. 8 shows one example of the *Armadillo* surface mesh after the different steps, with a zoom on the interior of the 3D mesh in Fig. 9 at all stages of the process.

5.1.1 Step 1: Generation of an Adaptive Hexahedral Mesh

In our experiment, the first step involves generating an adaptive hexahedral mesh. We achieve this using the algorithm outlined in Algo. 5.1. A sketch of this algorithm is presented here, but interested readers will refer to [DGLZD18] for all the details about message passing and how consistency of critical cells is guaranteed between the different blocks.

The main principle behind generating an adaptive hexahedral distributed mesh is to start with an initial grid of hexahedra, distribute them in the different blocks, and then run Algo. 5.1 in parallel on each block. However, if the number of initial hexahedra is smaller than the number of blocks, some blocks may be initially empty, causing their processes to be stalled until the subsequent balancing step. Given a maximal subdivision level *lmax* and an *Oracle* that determines whether a given volume should be subdivided, the algorithm iterates through all volumes, subdivides those that satisfy the oracle, and removes any hexahedron lying outside the given surface. In this experiment,

Algorithm 5.1: Generation of an adaptive hexahedral mesh in block B_i.

Data: B_i: a block containing initial hexahedra;
$lmax$: the maximal subdivision level;
Oracle: that returns if a hexahedra should be subdivided.
Result: Generates an adaptive hexahedral mesh in block B_i.

```
1 for curlevel ← 0 to lmax do
2     foreach volume v from B_i do
3         if Oracle(v) then
4             Subdivide(v);
5     Barrier();
6     removeOutsideVolumes(curLevel+1);
7     Barrier();
8     Balance cells that satisfy the Oracle (Algo. 4.1);
```

we use an intersection test between the input surface mesh and the hexahedra, but any criterion can be used instead depending on the application, like a Region Of Interest or intersection with a function.

Thus, the only –but substantial– modification of the original algorithm of [DGLZD18] is the insertion of line 8 that authorizes the balancing of the cells satisfying the oracle using the algorithm introduced in the previous section. We use the cell balancing at the end of each subdivision level in order to balance the computation time of the next subdivision level between all the blocks. Note that at this step, the transferred volumes are geometrically hexahedral-looking elements, but they can be topologically complex depending of the level of subdivision of each face. The non-intersecting cells are ignored at the balancing step since they are not concerned by the subdivision. They only induce a local memory cost but no additional computation time. Note also that this cell balancing is an option that can be enabled or disabled at any step.

5.1.2 Step 2: Making the Mesh Conformal

The second step starts from the adaptive hexahedral mesh generated from step 1 and turns it into a conformal mesh, as summarized in Algo. 5.2. All non-hexahedral volumes (in a topological sense, i.e., those with at least one subdivided face) are replaced by a special configuration of conformal hexahedra, tetrahedra, prisms, and pyramids. The full set of 325 existing configuration patterns and their replacement targets are given in [JL22] and implemented as a query/replace 3D operation, similarly to [DN22]. This algorithm is run in parallel for each block.

Algorithm 5.2: Modification of the mesh in block B_i to become conformal.

Data: B_i: a block containing an adaptive hexahedral mesh.
Result: Modifies the mesh to become conformal.

```
1 foreach volume v of B_i do
2     if v is not a hexahedron then
3         transform v into its compatible pattern;
```

5.1.3 Step 3: Projection of External Points on the Surface

The last step of our mesh generation process, presented in Algo. 5.3, starts with the mesh generated in step 2 and projects all exterior points of the mesh onto the input surface mesh. However, this projection may lead to some degenerated or inverted volumes. Such volumes are identified and removed from the mesh afterward. Once again, this algorithm is run in parallel for each block.

5.2 Overview of Experiments

We conducted five experiments to demonstrate the benefits of the cell transfer operation. Most of our experiments are based on multi-threaded (MT) mesh generation, while one is based on distributed mesh generation (MPI).

- The MT experiment in Sect. 5.3 addresses the performance improvement in the generation times for meshes with a similar number of final elements.

Algorithm 5.3: Projection of the external points of the mesh in block B_i on the surface.

Data: B_i: a block containing an adaptive hexahedral mesh;
S: a surface.
Result: Project the external points of the mesh on the surface.

```
foreach point p of B_i do
    if p is outside the surface then
        project p on S;
foreach volume v of B_i do
    if v is degenerated or inverted then
        remove v;
```

- The MT experiment in Sect. 5.4 explores the impact of the cell transfer strategy on the number of critical darts and the number of messages exchanged during the mesh generation process.
- The MT experiment in Sect. 5.5 illustrates the benefits of cell transfer to equilibrate computation load when the subdivision is localized within a region of interest (ROI).
- The MPI experiments in Sect. 5.6 focus on the better use of memory to reach more refined distributed meshes.
- The single core experiment in Sect. 5.7 shows that our method is competitive compared to a recent state-of-the-art octree method.

5.3 Impact of the Cell Balancing on Computation Times

All experiments in this section where run on an Intel® i9-10900K CPU @3.70GHz with 20 cores and 64 GB RAM.

In our first result, we compared the time required to complete each step of the mesh generation process, as well as the overall execution time, without and with cell balancing. For this, we implemented the two strategies S_1 and S_2 detailed in Sect. 4.

The computation times for the 4 input surface meshes are summarized in the different tables. For each step of the generation (Sect. 5.1) using 8 subdivision levels at maximum, we detail the computation times (in seconds), (W)ithout and with cell balancing using first excess block strategy S_1 and the minimization of critical cells strategy S_2. For cell balancing versions, we also give the speedup compared to (W)ithout cell balancing. For step 1 in Table 1, we can observe that the speedup when enabling cell balancing is on average 1.27 for S_1, and 1.3 using S_2.

	W (s)	S_1 (s, .)		S_2 (s, .)	
Armadillo	28.51	24.17	1.18	23.90	1.19
Camel	29.88	22.69	1.32	22.53	1.33
Happy	42.57	33.28	1.28	32.47	1.31
Octopus	36.03	27.29	1.32	26.66	1.35
Mean	34.25	26.86	**1.27**	26.39	**1.30**

Table 1: Step 1: computation times and speedup.

	W (s)	S_1 (s, .)		S_2 (s, .)	
Armadillo	71.50	50.04	1.43	50.04	1.43
Camel	74.84	48.38	1.55	48.06	1.56
Happy	101.01	59.26	1.70	59.29	1.70
Octopus	79.13	50.22	1.58	49.76	1.59
Mean	81.62	51.98	**1.56**	51.79	**1.57**

Table 2: Step 2: computation times and speedup.

	W (s)	S_1 (s, .)		S_2 (s, .)	
Armadillo	6.34	6.25	1.01	6.00	1.06
Camel	8.01	5.42	1.48	6.28	1.28
Happy	9.97	8.92	1.12	9.22	1.08
Octopus	8.08	7.65	1.06	7.90	1.02
Mean	8.10	7.06	**1.17**	7.35	**1.11**

Table 3: Step 3: computation times and speedup.

Better speedups are reached for step 2, as we can see in Table 2: about 1.56 for the first balancing method, and 1.57 for the second one. Pattern substitution is more demanding computationally, and we observe an even better impact of balancing.

For step 3, 1.17 and 1.11 speedup are respectively reached for each strategy (Table 3), being less significative as this step is the fastest of the three.

If we sum up the computation times of all the steps, we can observe in Table 4 that, in average, a 1.44 speedup is achieved when enabling cell balancing, for both balancing strategies. This demonstrates the interest of this option which allows to greatly reduces the global computation time of the parallel mesh generation method (from

	W (s)	S_1 (s, .)		S_2 (s, .)	
Armadillo	105.46	79.55	1.33	79.89	1.32
Camel	109.90	76.49	1.44	76.73	1.43
Happy	153.56	100.34	1.53	99.75	1.54
Octopus	123.24	83.47	1.48	82.28	1.50
Mean	123.04	84.96	**1.44**	84.66	**1.45**

Table 4: Global process (all the steps): computation times and speedup.

123s to 85s in average for our examples). The overhead implied by the message exchange for the cell transfer is fully compensated by the gain obtained by the cells transfer and balance of the computations between the different blocks.

5.4 Impact of the Cell Balancing on Critical Cells

In this section, we use the same experience and computer setup as in the previous section, but we study a different aspect.

Both versions (baseline and minimizing the number of critical cells) give similar final meshes and similar computational times. But, on average, as we can see in Table 5, the resulting number of critical darts is divided by two using strategy S_2, using the same experiments as in Section 5.3.

	S_1	S_2
Armadillo	2,689,962	1,366,102
Camel	2,608,704	1,257,068
Happy	3,730,332	1,835,318
Octopus	2,475,830	1,363,330
Mean	**2,876,207**	**1,455,455**

Table 5: Number of critical darts for the two cell balancing strategies: baseline (S_1) and minimizing critical cells (S_2).

With fewer critical cells, the amount of data to be stored in the associative arrays is reduced, resulting in less resident memory. In addition, this reduction of critical cells also decreases the number of exchanged messages between the different blocks. Indeed, when a critical cell is involved in an operation, a message must be sent to the other block that contains this cell to maintain the global consistency of the mesh. This is highlighted in Table 6, showing the total number of messages exchanged between all the blocks for the two different strategies. Again, on average, the number of messages sent is halved when using strategy S_2 compared to strategy S_1.

	S_1	S_2
Armadillo	1,501,062	755,694
Camel	1,480,613	714,668
Happy	2,077,958	995,862
Octopus	1,404,801	772,406
Mean	**1,616,109**	**809,658**

Table 6: Number of messages exchanged during the whole process of mesh creation (step 1, 2 and 3) for the two cell balancing strategies: baseline (S_1) and minimizing critical cells (S_2).

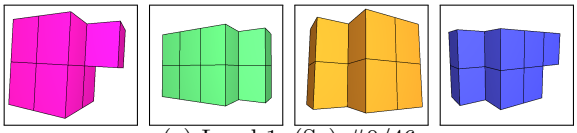
As mentioned previously, our method is very versatile and any balancing strategy could be used. The only thing one has to keep in mind when integrating a new strategy, being local or global, is that simple criteria may induce many critical cells, at the detriment of compactness. In our experiments, transfer and waiting time were not significant compared to mesh editing operations, but this could not always be the case, especially when network is slow.

To better understand the impact of each strategy on compactness, the *Armadillo* was subdivided into 4 blocks. It was found that Strategy S_2 produced more compact regions (as shown in Fig. 10) compared to the baseline strategy S_1. With S_1, small isolated volumes started appearing at level 5 and this increased significantly by level 7 (Fig. 11). However, it is important to note that the same issue could also arise with S_2 due to the second part of the strategy, which transfers a volume to any other block in deficit if no neighboring block accepts new volumes. While this constraint can be released to preserve compactness, it would lead to less optimal balancing.

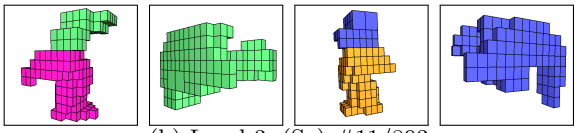
5.5 Impact of the Cell Balancing when using ROI

In this section, we used another configuration with more cores and another architecture: an AMD® Ryzen threadripper 2990wx @3GHz CPU, 32x2 logical cores and 128 GB RAM.

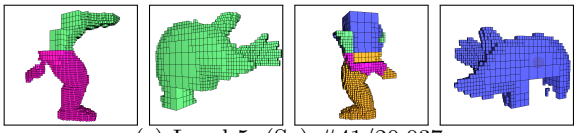
In our second experiment, we consider a cube and we refine it in a single corner (during step 1), causing subdivision to always occurs in the same block when balancing is inactive. This extreme



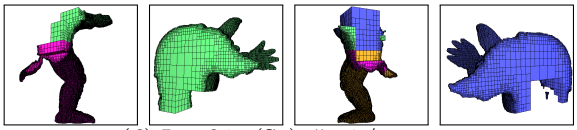
(a) Level 1, (S_2) #0/46



(b) Level 3, (S_2) #11/893

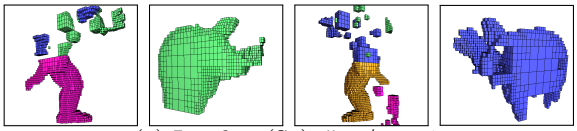


(c) Level 5, (S_2) #41/20,037

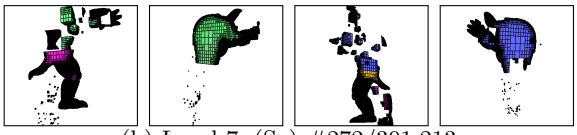


(d) Level 7, (S_2) #207/391,213

Fig. 10: Cell transfer **strategy S_2**. Level and corresponding number of transferred cells / total number of cells.



(a) Level 5, (S_1) #96/20,037



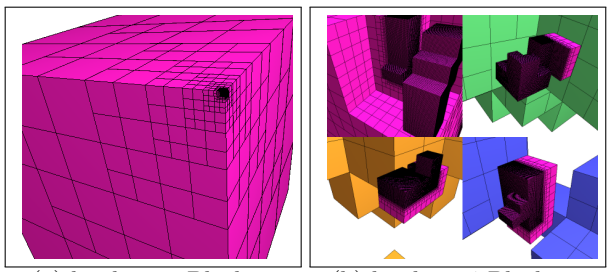
(b) Level 7, (S_1) #272/391,213

Fig. 11: Cell transfer **strategy S_1**. Level and corresponding number of transferred cells / total number of cells.

case illustrates the benefits of cell transfer to equilibrate computation loads. Such situation could arise in cases of localized contact between objects.

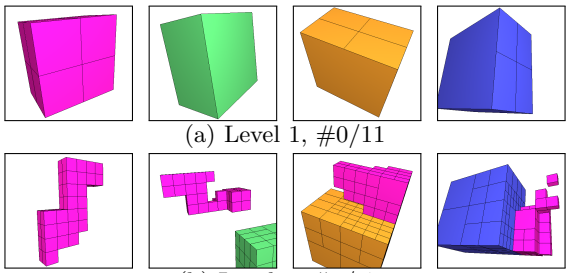
As expected, using 4 blocks (one per thread), the cells lying in the Region of Interest (RoI) are evenly distributed across the 4 blocks. This can be seen on Figure 12b, where the color of each cell indicates its originating block.

The same kind of experiment is performed setting a RoI on the head of the *Camel* model (Fig. 14a for a complete view). The evolution of the subdivision at different levels and how the

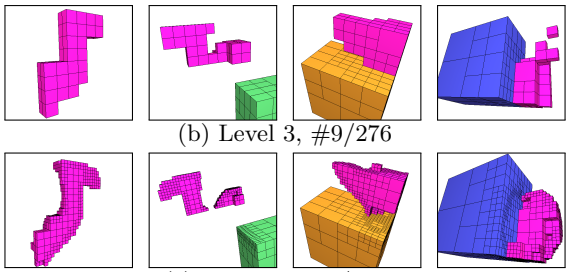


(a) level 12, 1 Block (b) level 12, 4 Blocks

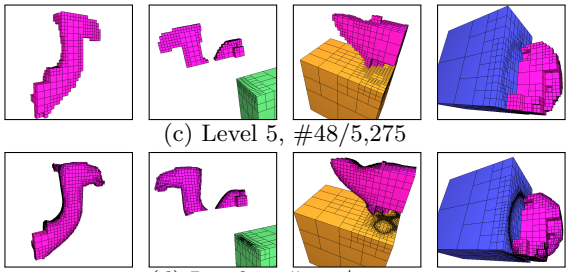
Fig. 12: Illustration of the volume cell transfer, balancing cells in the RoI.



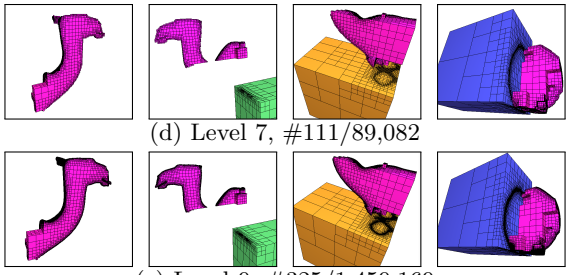
(a) Level 1, #0/11



(b) Level 3, #9/276



(c) Level 5, #48/5,275



(d) Level 7, #111/89,082

(e) Level 9, #325/1,459,169

Fig. 13: Illustration of the cell transfer, balancing all the cells between blocks (S_2). Level and corresponding number of transferred cells / total number of cells.

cells, all originated from Block B_0 (in pink), are distributed to other cells is shown Figure 13. Note that the transferred volume cells against the total number of volume cells ratio is decreasing. This is normal, as transferred cells still remain in the RoI and will be refined also in their destination block.

To be complete, we also performed a Strong Scalability test on the same experiment, subdividing the *Camel* in the RoI until Level 10, leading to highly unbalanced loads as mesh refinement only occurs in this area. Increasing the number of blocks (*ie.* threads), a speedup of ≈ 12 is quickly reached (Fig. 14b) for a relatively small number of threads.

In this extreme case, the speedup is leveled by the structural handling of the critical cells when the number of blocks goes greater. According to Amdahl's Law that can be formulated as [Amd67]:

$$speedup = 1/(s + p/N)$$

where s and $p = 1 - s$ are the respective proportion of execution time spent on the serial and parallelized parts, and N is the number of processors; $s \approx 0,07$ meaning that more than 93% of our program still runs in parallel for 64 threads. This demonstrates the good scalability of our volume transfer operation.

5.6 Impact of the Cell Balancing on the Memory Usage

For this experiment, our program was run on a High-Performance Computing (HPC) platform designed for parallel computing using MPI. The platform comprises 16 servers interconnected through InfiniBand, each equipped with 32 cores and 128 GB of RAM. The model of each server is an Intel® Xeon® CPU E5-2698 v3 @2.30GHz (2 sockets with 16 cores per socket). However, to allow resource sharing with other users, our experiment was restricted to utilizing only 8 servers.

	W (#)	S_2 (#)
Armadillo	60,814,077	123,165,351
Camel	112,591,944	248,917,966
Happy	46,834,266	165,174,016
Octopus	129,062,620	203,624,153
Mean	**87,325,727**	**185,220,372**

Table 7: Number of volume elements for the MPI experiment (W)ithout balancing and with cell balancing strategy S_2.

We carried out an experience of generating hexahedral meshes with the most volume elements using available resources (256 cores), one without cell balancing (W), and the other with cell balancing strategy S_2. We set the *lmax* parameter to 10 for W and S_2 to make sure the W runs terminate. We also set a *max-memory* parameter to 475,000 KB to prevent each core from exceeding its allocated RAM as each core has its own RAM budget on the HPC platform. The subdivision stops in a process when it has reached this limit. Table 7 shows that the load balancing affords to reach more refined meshes. Indeed, by repartitioning and thus balancing volumes among the different processors, each processor can maximize its number of volumes to subdivide according to its available RAM, while, without repartitioning, some cores are under exploited, explaining thus the smaller number of hexahedra at the end.

	W (s)	S_2 (s, .)	
Armadillo	254.32	116.21	2.19
Camel	273.76	198.81	1.38
Happy	236.70	137.28	1.72
Octopus	252.21	203.71	1.24
Mean	254.25	164.00	**1.63**

Table 8: Global process (all the steps): computation times, in seconds, (W)ithout and with cell balancing strategy S_2, including speedup compared with no cell balancing.

On average, the cell balancing resulted in a speedup of about 1.63, as shown in Table 8.

5.7 Comparison with state-of-the-art methods

As presented in Section 2, many works have been proposed for mesh generation and editing, distributed or not. Unfortunately, few are Open Source[1]. In the context of our work, reliable comparison with distributed approaches is not straightforward. They are most of the time too specialized, relying on a single type of element (either tetrahedron or hexahedron) or optimized for a specific application, making them difficult to evolve for handling the generic meshes considered here.

For comparison purposes, we selected a mixed-element mesh generation and refinement approach

[1] Our software is publicly available https://hal.science/hal-04650145v1.

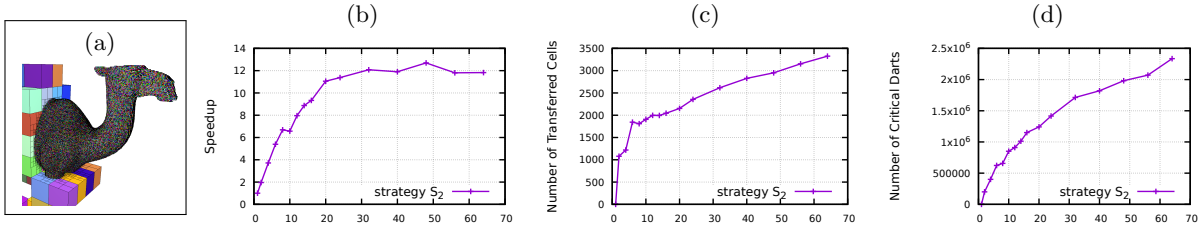


Fig. 14: Strong scalability, increasing the number of threads, on **Camel** subdivision Level 10 in RoI, resulting in #14,235,373 cells (step 1, strategy S_2). (b) Speedup (Total time **With No** transfer/**With** transfer); (c) Number of Transferred Cells and (d) Number of Critical Darts.

very similar to our proposal [JL22], although not distributed. We present informative comparisons using the *Camel* model in Figure 15 on a log-log scale. To ensure fairness, we started with a single cell to avoid any initial grid variations, using a single block on a single processor.

Both methods are overall slightly similar (Fig. 15) but at a higher Resident Memory cost for our method. Going into the details, we can notice that Step 1 is faster, probably due to the fact that the Combinatorial Map behaves better during the balance process than the haddock structure used in [JL22] to circumvent the poor neighborhood handling offered by octree-like structures. However, Step 2 is relatively demanding, involving costly topological operations of query and replace (un-sewing and sewing cells). These operations are implemented in a generic way, and can be specialized and optimized. Step 3 is less demanding than the two other steps, and comparison between our simple point projection against a more complete process of surface pattern handling is made difficult.

Overall, our proposal demonstrates that a topological framework can be competitive with a popular octree-like structure. Furthermore, the parallel, multi-threaded, or MPI versions of our method provide a significant advantage over the method described in [JL22].

6 Conclusion

In this paper, we introduced a novel volume transfer operation based on a distributed combinatorial map. For the first time, this operation is defined for cells with any topology. It can be used to dynamically balance the computation loads during the generation or editing of large meshes. We developed this solution within a framework for generating mixed-element meshes using volume subdivision patterns and surface fitting. During the generation process, we can also constrain one level of subdivision between neighboring cells, facilitating the generation of a conformal mesh at the end. In general terms, this is a challenging problem (for example, retrieving neighbors after subdivision is not trivial in octree-like meshes), and we proposed a generic solution for handling critical cells and tracking of neighboring blocks.

Our multi-threaded and MPI implementations demonstrate the effectiveness of this transfer operation in reducing total subdivision time and achieving finer mesh resolution.

Besides its proven efficiency, one crucial point of our approach is its complete genericity, being able to handle and transfer any type of volume cell topology. In the search for an optimal trade-off between the cost of the cell transfer and the gain in computation time or memory, the user can freely choose to activate the transfer volume at any step and modify the transfer decision criteria as needed. This is in contrast to most other state-of-the-art methods, which are typically optimized for a specific goal and designed for use with a single type of element (usually tetrahedra or hexahedra). Our method's genericity allows us to deal with mixed-element meshes with no restriction on the topology, as demonstrated in the different steps of the experiments. So, our proposal could benefit any application to dynamically balance the cells of a mesh when necessary.

Dynamic repartitioning is a powerful tool for optimizing execution speed, as it can significantly improve performance for any operation that must be performed on meshes. This can be easily achieved by transferring volumes between blocks.

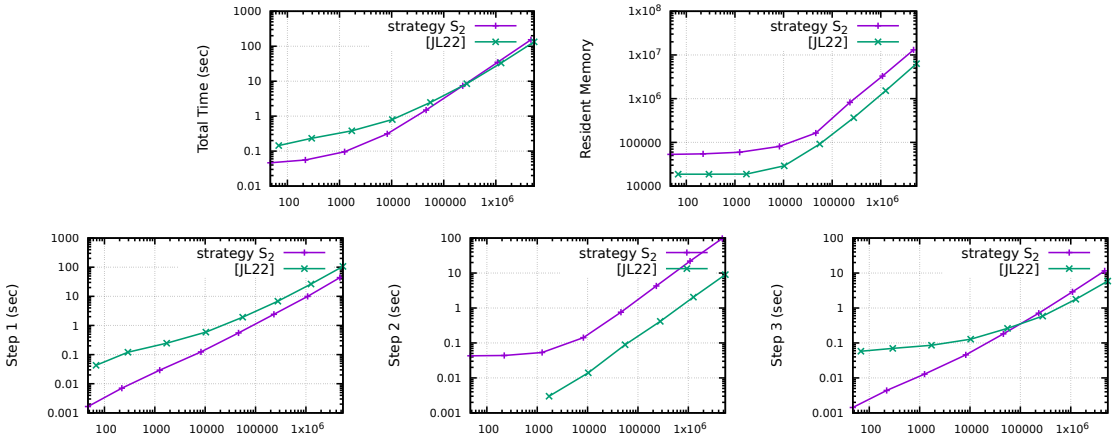


Fig. 15: Comparison with [JL22]. For the *Camel* model, the different plots show Total and Partial Times or Resident Memory according to the final number of Cells, at a log-log scale.

Then, diverse strategies can be set and tuned according to criteria depending on the application, such as improving speed, compactness, or minimizing interfaces, etc. A positive side effect of volume transfer is that the results are more resistant to the initial grid configuration. With fewer initial filled blocks than available threads, and with no transfer, an empty block at the beginning will remain so until the end. By contrast, starting with a single volume and transferring new cells into empty blocks on the very next turn can result in a more even distribution of cells.

As a future work, we propose the simultaneous transfer of several volumes between two blocks. This extension would be useful for moving an interface between blocks, for example after a repartitioning phase. We expect fewer, but potentially longer `TR` messages, which may be compressed to reduce the overall transmission. We also plan to develop other operations for distributed 3-maps, such as local refinement or de-refinement of cells, merging cells in a given area, or topological cutting of a part of the mesh by a plane. All these future work are made possible thanks to the use of distributed 3-maps, and will be simplified and optimized by the use of our generic volume transfer operation.

Acknowledgments

We gratefully acknowledge support from the CNRS/IN2P3 Computing Center (Lyon - France) for providing computing and data-processing resources needed for the MPI experiments.

References

[ALSS06] Frédéric Alauzet, Xiangrong Li, E. Seegyoung Seol, and Mark S. Shephard. Parallel anisotropic 3d mesh adaptation by mesh modification. *Eng. with Comput.*, 21(3):247–258, may 2006.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67 (Spring), page 483–485. ACM, 1967.

[BBHK12] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software*, 38, jan 2012.

[BGG+08] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Georg Stadler, Eh Tan, Tiankai Tu, Lucas C. Wilcox, and Shijie Zhong. *Scalable Adaptive Mantle Convection Simulation on Petascale Supercomputers*. IEEE, 2008.

[BGG+10] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Georg Stadler, Tim Warburton, and Lucas C. Wilcox. Extreme-scale amr. 2010.

[BWG11] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. P4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33:1103–1133, 2011.

[CA15] Daniela Cabiddu and Marco Attene. Large mesh simplification for distributed environments. *Comput. Graph.*, 51(C):81–89, oct 2015.

[CDF+03] Paul M. Campbell, Karen D. Devine, Joseph E. Flaherty, Luis G. Gervasio, and James D. Teresco. Dynamic octree load balancing using space-filling curves. Technical report, Technical Report CS-03-01, Williams College Department of Computer Science, 2003.

[Dam11] G. Damiand. Combinatorial maps. In *CGAL User and Reference Manual*. 3.9 edition, 2011. http://www.cgal.org/Pkg/CombinatorialMaps.

[DGLZD18] G. Damiand, A. Gonzalez-Lorenzo, F. Zara, and F. Dupont. Distributed combinatorial maps for parallel mesh processing. *Algorithms*, 11(7), August 2018.

[DHF+09] William Dawes, Simon Harvey, Simon Fellows, Neil Eccles, D Jaeggi, and Will Kellar. *A Practical Demonstration of Scalable, Parallel Mesh Generation*. 2009.

[DL14] G. Damiand and P. Lienhardt. *Combinatorial Maps: Efficient Data Structures for Computer Graphics and Image Processing*. A K Peters/CRC Press, September 2014.

[DN22] G. Damiand and V. Nivoliers. Query-replace operations for topologically controlled 3d mesh editing. *Computers & Graphics (C&G)*, 106:187–199, August 2022.

[DSS17] Gerrett Diamond, Cameron W. Smith, and Mark S. Shephard. Dynamic load balancing of massively parallel unstructured meshes. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, New York, NY, USA, 2017. Association for Computing Machinery.

[FANF12] Rosa Filgueira, Malcolm Atkinson, Alberto Nuñez, and Javier Fernández. An adaptive, scalable, and portable technique for speeding up mpi-based applications. In *Euro-Par 2012 Parallel Processing*, pages 729–740. Springer Berlin Heidelberg, 2012.

[FLS+97] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws. *Journal of Parallel and Distributed Computing*, 47(2):139–152, 1997.

[HBK+23] Johannes Holke, Carsten Burstedde, David Knapp, Lukas Dreyer, Sandro Elsweijer, Veli Ünlü, Johannes Markert, Ioannis Lilikakis, Niklas Böing, Prasanna Ponnusamy, et al. t8code v. 1.0-modular adaptive mesh refinement in the exascale era. In *SIAM International Meshing Roundtable Workshop 2023 (SIAM IMR 2023)*, Amsterdam, Netherlands, 2023.

[HD00] Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. *Computer methods in applied mechanics and engineering*, 184(2-4):485–500, 2000.

[Hen00] Bruce Hendrickson. Load balancing fictions, falsehoods and fallacies. *Applied Mathematical Modelling*, 25(2):99–108, 2000.

[HKB21] Johannes Holke, David Knapp, and Carsten Burstedde. An optimized, parallel computation of the ghost layer for adaptive hybrid forest meshes. *SIAM Journal on Scientific Computing*, 43(6):C359–C385, 2021.

[HS18] Jaber J. Hasbestan and Inanc Senocak. Binarized-octree generation for cartesian adaptive mesh refinement around immersed geometries. *J. Comput. Phys.*, 368(C):179–195, sep 2018.

[JL22] F. Jaillet and C. Lobos. Fast quadtree/octree adaptive meshing and re-meshing with linear mixed elements. *Engineering with Computers*, 38(4):3399–3416, 2022.

[JLY10] Hua Ji, Fue Sang Lien, and Eugene Yee. A new adaptive mesh refinement data structure with an application to detonation. *Journal of Computational Physics*, 229:8981–8993, 2010.

[KK93] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[LB10] Justin Luitjens and Martin Berzins. Improving the performance of uintah: A large-scale adaptive meshing computational framework. In *Int. Symp. on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[LCW+06] Orion S Lawlor, Sayantan Chakravorty, Terry L Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant V Kalé. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22:215–235, 2006.

[Lie94] P. Lienhardt. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *Int. J. Comput. Geometry Appl.*, 4(3):275–324, 1994.

[LLM+12] Akhil Langer, Jonathan Lifflander, Phil Miller, Kuo-Chuan Pan, Laxmikant V Kale, and Paul Ricker. Scalable algorithms for distributed-memory adaptive mesh refinement. In *24th Int. Symp. on Computer Architecture and High Performance Computing*, pages 100–107. IEEE, 2012.

[LMA15] Adrien Loseille, Victorien Menier, and Frédéric Alauzet. Parallel generation of large-size adapted meshes. *Procedia Engineering*, 124:57–69, 2015. 24th International Meshing Roundtable.

[LPC21] Marco Livesu, Luca Pitzalis, and Gianmarco Cherchi. Optimal dual schemes for adaptive grid based hexmeshing. *ACM Trans. Graph.*, 41(2), dec 2021.

[MPR19] Célestin Marot, Jeanne Pellerin, and Jean-François Remacle. One machine, one minute, three billion tetrahedra. *International Journal for Numerical Methods in Engineering*, 117(9):967–990, 2019.

[NDB17] Tuan T. Nguyen, Vedrana A. Dahl, and J. Andreas Bærentzen. Cachemesh, a dynamics data structure for performance optimization. *Procedia Engineering*, 203:193–205, 2017.

[PCS+22] Nico Pietroni, Marcel Campen, Alla Sheffer, Gianmarco Cherchi, David Bommes, Xifeng Gao, Riccardo Scateni, Franck Ledoux, Jean

Remacle, and Marco Livesu. Hex-mesh generation and processing: A survey. *ACM Trans. Graph.*, 42(2), oct 2022.

[SB10] Rahul S. Sampath and George Biros. A parallel geometric multigrid method for finite elements on octree meshes. *SIAM Journal on Scientific Computing*, 32:1361–1392, 2010.

[SS06] E. Seegyoung Seol and Mark S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 22:197–213, 12 2006.

[TCL+20] Jing Tang, Pengcheng Cui, Bin Li, Yaobin Zhang, and Hang Si. Parallel hybrid mesh adaptation by refinement and coarsening. *Graphical Models*, 111, 9 2020.

[Wei85] K. Weiler. Edge-based data structures for solid modelling in curved-surface environments. *Computer Graphics and Applications*, 5(1):21–40, 1985.