Engineering with Computers (2016) 32:655–674
DOI 10.1007/s00366-016-0444-3

ORIGINAL ARTICLE

Parallel generation of meshes with cracks using binary spatial decomposition

Markos O. Freitas[1] · Paul A. Wawrzynek[3] · Joaquim B. Cavalcante-Neto[1] · Creto A. Vidal[1] · Bruce J. Carter[3] · Luiz F. Martha[2] · Anthony R. Ingraffea[3]

Received: 19 August 2015 / Accepted: 3 March 2016 / Published online: 16 March 2016

Abstract This work describes a technique to generate tetrahedral meshes with cracks using parallel computers with distributed memory. This technique can be used for models without cracks as well. It employs a binary partitioning structure that uses axis-aligned planes to decompose the domain. Those decomposing planes are determined based on a refined octree that is built to estimate the amount of work necessary to generate the whole mesh, so that the amount of work in each subdomain is approximately the same. A serial advancing front technique is used in each subdomain concurrently, in such a way that the generated tetrahedra do not cross the decomposing planes. After local synchronizations, meshes are generated interfacing the subdomains. The results show that the prediction of the number of elements in each subdomain is accurate, leading to a well-balanced algorithm and to a good speed-up. Also, the meshes generated in parallel have very good quality, similar to the that of a serially generated mesh.

Keywords Parallel mesh generation · Advancing front technique · Binary spatial partitioning · Cracks

✉ Markos O. Freitas
markos@lia.ufc.br

Paul A. Wawrzynek
wash@fac.cfg.cornell.edu

Joaquim B. Cavalcante-Neto
joaquimb@lia.ufc.br

Creto A. Vidal
cvidal@lia.ufc.br

Bruce J. Carter
bjc21@cornell.edu

Luiz F. Martha
lfm@tecgraf.puc-rio.br

Anthony R. Ingraffea
ari1@cornell.edu

1 Universidade Federal do Ceará (UFC), Fortaleza, Brazil

2 Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil

3 Cornell University, Ithaca, USA

1 Introduction

This work presents a parallel technique for generating three-dimensional tetrahedral meshes by the advancing front method. The technique was designed to meet eight requirements: to respect the input front, discretized in triangular faces, i.e., no boundary refinement can be employed; to produce well-shaped elements, avoiding elements with poor aspect ratios; to provide good transitions from refined to coarse regions of the mesh; to respect cracks given as input; to be independent of the multiprocessing memory architecture (shared or distributed) and of the domain's dimension (2 or 3); to present a good estimation of the number of elements generated in each subdomain; to present a good load balancing, distributing the subdomains among the processing entities uniformly; to present a good speed-up, as a consequence of the two previous requirements. The algorithm is based on a serial 3D advancing front strategy developed by some of the authors [1–3].

The first four requirements were taken into account in the development of the serial algorithm and, therefore, must be satisfied by the parallel algorithm as well. The first requirement is very important in many problems, such as those encountered in simulations in which the domain contains regions with different materials and holes. In those situations, it is often desirable that the mesh conform to the region's existing boundary discretization. Regarding the second requirement, although the technique does not guarantee bounds on element aspect ratios, care is taken at each step to generate elements with the best possible shapes. Concerning the third requirement, in many applications, the size difference between elements in a refined region and those in a coarse region is larger than two orders of magnitude. Thus, to provide good transition capabilities is an important requirement in practice. The fourth requirement is important in crack propagation simulations, because elements on one side of a crack cannot be directly connected to the elements on the other side. In this method, the crack surface is represented as a null area region, made of input faces geometrically coincident, but whose normals have opposite directions. Thus, no preprocessing step is needed to identify which faces or vertices belong to cracks. An existing crack belongs to an individual region to be meshed. The crack is either completely inside the region or it is in contact with the region's boundary surface. Although the main focus of this paper is on a parallel mesh generation method, which was designed to deal trivially with crack faces and vertices, it does work on models without cracks as well, with no modifications.

The last four requirements concern only the parallel algorithm. It is desirable that it runs on several types of current parallel platforms, ranging from multi-core desktops to cluster computers. Therefore, the algorithm must be generic to encompass shared and distributed-memory architectures, and the implementation must reflect this issue. Although the results shown in this work were obtained with a distributed-memory machine, the algorithm is really independent of the multiprocessing memory architecture. This is an advantage of this technique, since many of the parallel techniques presented in the literature work only for one specific type of multiprocessing memory architecture. Also, the algorithm works for both two and three dimensions, i.e., it can be used to generate triangular as well as tetrahedral meshes, even though the 3D version is the focus of this work. The last three requirements are closely related to one another. The amount of work (called load in high performance computing) is well estimated and the partitioning is appropriate to decompose the domain in subdomains with approximately the same load, resulting in good speedup.

The remainder of this work is divided into four sections. Sect. 2 describes the related work. Sect. 3 describes the devised parallel technique. Section 4 shows the tests and results. Section 5 presents the conclusions and some recommendations for future work.

2 Related work

Several works in the literature use the parallel algorithm model of recurrently subdividing the subdomains in half as much as possible [4–12]. That model is also used in our work. The main differences among those works consist in: how to position the decomposing plane (or line, in 2D); how to generate the mesh in this decomposing plane, which is usually done prior to the generation of the mesh inside the subdomains; and how to generate the mesh in each subdomain, usually done by an advancing front technique (AFT) or by a Delaunay mesh generator.

Most ideas on how to divide the subdomains try to balance the physical or geometrical properties on both sides of the dividing plane. In [4, 5], lines, based on inertial axes, decompose the 2D subdomains. In [6–10], planes are used to decompose the 3D domain, according to the number of faces, length of edges, global axes or inertial axes. When a subdomain is cut by a plane, a 2D intersection region is the common interface between the two halves located on each side of the plane. One possible idea of ensuring conformity of the meshes that are generated on both sides of the plane is to generate, a priori, a mesh in the interface region. In [6–10], the interfacing mesh is constructed based on a Delaunay triangulation. In [11], the plane decomposing the domain is placed on the center of density, given by an octree. The interfacing mesh is composed of tetrahedra, generated by advancing the faces that cross the decomposing plane. In [12], a kd-tree is built on blocks that decompose the domain. Delaunay meshes are generated in each block, and the interfacing meshes are generated as the algorithm climbs up the tree structure.

The same idea of subdividing the domain in half is used in several graph/mesh partitioning techniques for finite element meshes [13–15]. Those techniques are also used in mesh generation when a background mesh is available for decomposition, such as in [4, 9, 16–22]. Other works consider each element of the background mesh as a subdomain for mesh generation, such as [22–25]. The techniques that use elements or groupings of elements of a background mesh to decompose the domain are classified as discrete domain decomposition (DDD) techniques in [26]. In those techniques, after determining the subdomains, their interfacing meshes, as well as the original surface mesh, are refined and each subdomain is meshed independently.

Techniques that do not use a background mesh are classified as continuous domain decomposition (CDD) techniques in the same work, and can use any method to decompose the domain. Therefore, in addition to recursive binary subdivision [4–12], as mentioned before, blocks [27–30], recursive spatial decompositions (quadtrees or octrees) [31–34], spatial sorting [35], and data structure partitioning [36] are also among the possible choices. As pointed out in [26], techniques that generate the interfacing mesh a priori might create artifacts in the interior of the mesh that degrade its final quality. This can happen both with CDD as well as with DDD techniques.

Several parallel meshing techniques have been proposed in the literature: 2D Delaunay techniques [27–29, 31, 35, 36]; 3D Delaunay technique [30]; combination of AFT with template meshing [32]; AFT [21, 33, 34]. Those AFT-based parallel techniques advance faces so that they do not cross the subdomain's limits, defined by either the leaf-cells of a quadtree [34], the leaf-cells of an octree [32, 33], or the partitioning of a mesh [21]. After that, to generate the interfacing meshes, several strategies have been proposed: hierarchical front repartitioning [32]; subdomain shifts [33, 34]; and graph-coloring scheme to determine the communication patterns [21]. Since the interfacing meshes are generated after the generation of the meshes in the subdomains, these techniques are called a posteriori.

The work presented here shares similarities with some of the aforementioned works. It subdivides the domain in half, such as in [4–12]. While the work in [12] uses a kd-tree, the work presented here uses a binary spatial partitioning (BSP) data structure, such as in [9], which can be seen as a generalization of a kd-tree. Similar to what is done in [11, 34], a quadtree/octree is used to estimated the load, but, in this work, the positioning of the decomposing plane is performed differently. Unlike the strategy used in [4–11], the interfacing meshes are generated a posteriori, similarly to what is done in [12, 21, 32–34]. Like in [12], the interfacing meshes are generated as the algorithm climbs up the tree. Unlike what is done in [12] and similar to what is done in [11, 21, 33, 34], the meshes are generated using solely an advancing front technique. Additionally, none of the related works address all the requirements cited in Sect. 1, which is the main objective of this work, especially in effectively dealing with cracks.

3 Description of the parallel technique

The technique presented here works for two- and three-dimensional models. However, despite the focus being on 3D cases, 2D illustrations are used only for the sake of clarity. The technique receives as input a list of triangular faces describing one or more objects, which might have holes or cracks. This boundary representation, which is the initial advancing front, defines a domain that is decomposed by an axis-aligned binary spatial partitioning (BSP) tree, built in such a way that the amount of work (the load) in each subdomain (a leaf of the BSP tree) is approximately the same. The load is estimated using a fine octree that represents the density distribution over the domain.

The subdomains are meshed simultaneously, so that no tetrahedral element crosses the bounding limits of its subdomain. After that, meshes interfacing the subdomains are generated connecting two sibling leaves. This interface meshing is also performed connecting sibling internal nodes, as the BSP tree is traversed from the leaves to the root. When the root is reached, the mesh is completely generated, although it remains scattered among all the processes.

The use of a BSP can be justified by the combination of its simplicity with its flexibility. It is a binary tree, a data structure that is not very complex to implement, and the positioning of the plane that decomposes each internal node of the tree can be placed anywhere. In particular, this work shows how to position the plane such that the estimated workload is approximately divided in two, instead of dividing a geometric size, such as length, width or depth, which happens in other geometrically-based space decomposition data structures, such as an octree. One restriction of this work is that the planes decomposing the domain must be perpendicular to one of the global axes, due to the data structure used to estimate the workload. However, since any axis can be chosen in each step, even this restricted BSP implementation is still more flexible than a kd-tree data structure, which has a fixed order for choosing the axis.

3.1 Load estimation

In high performance computing (HPC), the load is a value that tries to predict the amount of work to be performed on a subdomain. In practice, this amount of work should reflect the execution time of a subdomain. However, to predict the runtime is a very difficult problem, because it involves external factors, such as memory access time, communication time, and execution of other processes. Therefore, instead of calculating the runtime, the complexity of the algorithm is taken into account.

Since the complexity of 3D mesh generation algorithms usually depends on the number of elements of the generated mesh, the load must be proportional to the number of generated elements. However, the number of elements will be known only after the mesh is fully generated and, thus, the load must be estimated using additional information. This work employs a fine octree as an additional structure to estimate the load. An octree is used because it can reflect the discretization level of the mesh inside the domain, as well as transitions between refined and coarse regions of the desired mesh. More details on how this octree is built can be found in [1–3].

After the load-estimation octree of a model is built, its leaf-cells are classified as inside, outside, or on the boundary of the model. The cells on the boundary are those that cross any of the model's faces. Once the "on the boundary" cells are found, their neighbors are classified using the unit normal vectors to the model's boundary faces, which point to the interior of the model by convention. Next, an

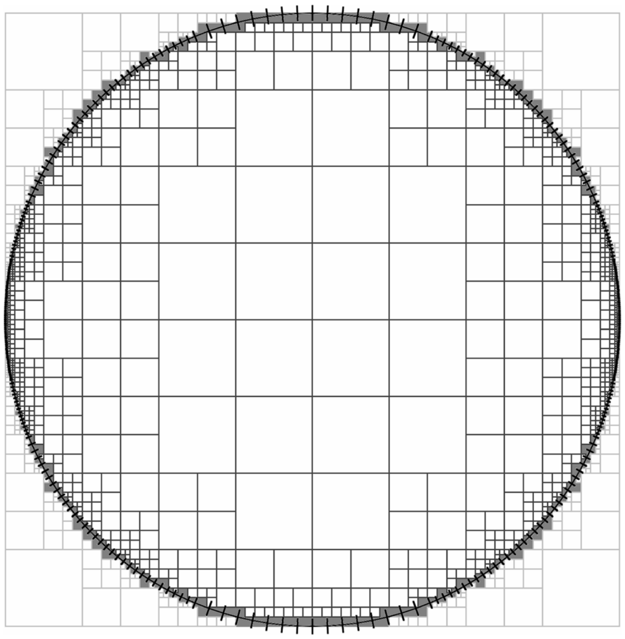

Fig. 1 Example of a load-estimation quadtree

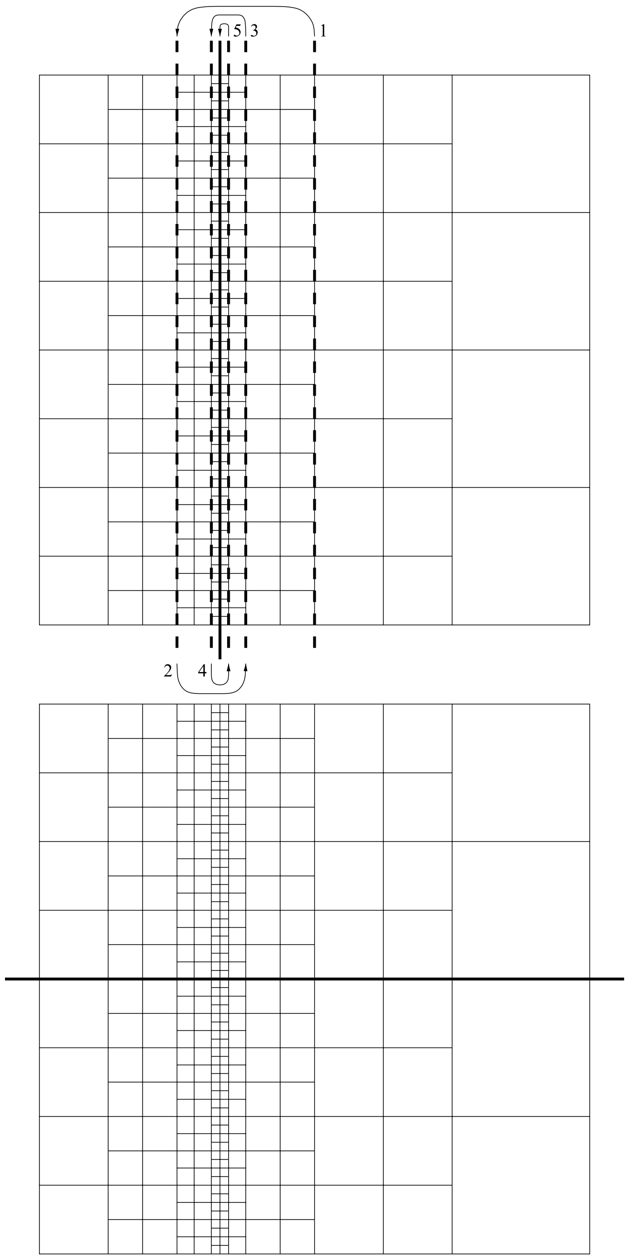

Fig. 2 Domain decompositions performed by the BSP, on the *X* axis (*top*) and on the *Y* axis (*bottom*). *Dashed lines* represent previous locations of the decomposition plane, which are displaced according to the *arrows*. The continuous line represents its final position

algorithm similar to a flood-fill [37] is employed to classify the remaining cells.

The total load estimation of the model is the number of cells classified as inside or on the boundary of the model domain, called full cells. In Fig. 1, a load-estimation quadtree is illustrated. Full cells, the ones considered for load estimation, are those inside the domain (cells with dark lines) and the ones on the boundary (filled cells). During the decomposition of the domain, full cells inside a subdomain are the ones used in its load estimation.

3.2 Domain decomposition

The idea for constructing the BSP is to perform a recursive binary search, over the octree, for the best location of the decomposing plane to subdivide a BSP cell, i.e., the location where the two new subdomains have approximately the same load estimation. This is performed for the three global axes, and the best one is chosen. To break ties, for each axis, the load of the interface connecting the two subdomains is also estimated, and the axis with the lightest interface is chosen.

3.2.1 Decomposition on an axis

Initially, a cube, equal to the root of the octree, is built as the root of the BSP. To subdivide this cell in two, an attempt is made to place the decomposing plane on the center of this cell, on the *X* axis. This plane splits the children of the octree's root, leaving 4 children on the left and the other four on the right in 3D (two children on each side, in 2D). Thus, each BSP cell corresponds to 4 children of the root of the octree.

If the loads of these BSP cells have the same value, this position is the best for the *X* axis. Otherwise, the heavier child is detected and the decomposition plane is moved to its center, on the same axis (Fig. 2, top). This new subdivision transfers some of the octree's cells from one BSP child to the other, thus decreasing the load estimation of the heavier cell and increasing the load estimation of the lighter cell.

That procedure is performed recursively, and has two stopping criteria: the loads of the two subdomains are equal; or a leaf of the octree is reached, i.e., it is no longer possible to subdivide. Among the several tested locations, the one that best distributes the load estimation between the subdomains is chosen, i.e., the one that minimizes the absolute difference between the two load estimations.

Figure 2 (top) depicts the procedure on a quadtree, considering that all its cells are classified as full cells. In the figure, the dashed lines represent previous locations of the decomposition plane, the continuous line represents its final position, and the arrows and the numbering show its displacement as the binary search descends the quadtree. The same procedure is also performed for the other axes (Fig. 2, bottom). In the same figure, on the Y axis, the initial position of the plane decomposes the domain in equal parts, due to its symmetry. Therefore, no displacement was required in that direction.

Notice that, if the decomposing plane is initially placed on the geometric center of a BSP cell, it is possible that its location does not match an octree division, and thus a full cell might be crossed. For example, in Fig. 2 (top), if the BSP cell on the left is further subdivided, the initial position would lie inside the second column of quadtree cells, not on their edges. To avoid such cases, the largest octree cells inside the BSP cell are found, their extreme coordinates in that axis are uniquely determined, and their median value is taken as initial position for the decomposing plane.

Furthermore, during the selection of the best decomposition among the axes, ties are broken using the load estimation of the interfaces. This load estimation is performed by finding one layer of octree cells adjacent to the decomposition plane, in each side of the plane, removing the load corresponding to these layers from the subdomains, and assigning those layers to the interface. Therefore, the load estimation in a subdomain is calculated as the number of full octree cells inside the domain minus one layer of octree cells adjacent to the partitioning planes around the subdomain. For example, in Fig. 2 (top), the load estimation for the left subdomain is 104, for the right subdomain, 84, and, for the interface, 128; in Fig. 2 (bottom), the load estimations for the top and bottom subdomains and for the interface are, respectively, 145, 145 and 26.

3.2.2 *Best decomposition among the axes*

One of the main purposes of this work is to balance the estimation of the load among the subdomains, using the estimation of the load on interfaces as tie breakers. However, since the interfacing mesh is generated a posteriori, the time to generate them can be crucial for the whole parallel technique. Therefore, a balance between minimizing the absolute difference between the load estimations of the subdomains and minimizing the load estimations of the interfaces must exist.

That balance, which, in a certain way, is analogous to balancing communication and computation in parallel algorithms in general, is controlled by a threshold $t \in [0, 1]$ in this work. That threshold controls the acceptance of a small imbalance between subdomains, as long as their interface is small. The balance between subdomains regards only one axis, while testing the size of the interface requires comparisons between two axes.

The balance (or unbalance) between the subdomains of an α-axis is calculated as

$$(b_{ij})_\alpha = \frac{|(\mathrm{LE}_i)_\alpha - (\mathrm{LE}_j)_\alpha|}{\max\{(\mathrm{LE}_i)_\alpha, (\mathrm{LE}_j)_\alpha\}},$$

where $(b_{ij})_\alpha$ is the balance factor between subdomains i and j defined by the α-axis, α is either X, Y or Z, and $(\mathrm{LE}_i)_\alpha$ and $(\mathrm{LE}_j)_\alpha$ are their respective load estimations for that axis. A perfect balance occurs when both subdomains have the same work load and the worst unbalance happens when one subdomain gets all the work load. Thus, if $(b_{ij})_\alpha \leq t$, where t is the given threshold, the α-axis is said to be relatively well-balanced:

$$(b_{ij})_\alpha \begin{cases} = 0 & \text{(perfectly balanced)} \\ = 1 & \text{(completely unbalanced)} \\ \leq t & \text{(relatively well-balanced).} \end{cases}$$

The comparison between interfaces is calculated as

$$(b_{ij})_{\alpha\beta} = \frac{(\mathrm{LE}_{ij})_\alpha}{(\mathrm{LE}_{ij})_\beta},$$

where the subscript ij indicates the interface between subdomains i and j, $(b_{ij})_{\alpha\beta}$ is the balance factor between the α-axis and the β-axis, α and β are either X, Y or Z, and $(\mathrm{LE}_{ij})_\alpha$ and $(\mathrm{LE}_{ij})_\beta$ are the load estimations for the interfaces in the α-axis and in the β-axis, respectively. Thus, the following cases occur:

$$(b_{ij})_{\alpha\beta} \begin{cases} \to 0 & \text{if } (\mathrm{LE}_{ij})_\alpha \ll (\mathrm{LE}_{ij})_\beta \\ \to \infty & \text{if } (\mathrm{LE}_{ij})_\alpha \gg (\mathrm{LE}_{ij})_\beta \\ = 1 & \text{if } (\mathrm{LE}_{ij})_\alpha = (\mathrm{LE}_{ij})_\beta \\ \leq t & \alpha \text{ is said do be much lighter than } \beta. \end{cases}$$

Table 1 Balance factors of decompositions in Fig. 2

Axis	Subdomains	Interfaces
X	$\frac{\lvert 104-84\rvert}{\max\{104,84\}} \cong 0.1923$	$\frac{128}{26} \cong 4.9230$
Y	$\frac{\lvert 145-145\rvert}{\max\{145,145\}} = 0.0000$	$\frac{26}{128} \cong 0.2031$

Table 1 shows the balance factors for the decompositions shown in Fig. 2.

When comparing two axes to select which one is better, three main tests are performed, one that considers the balancing factors and the threshold, one that considers only the balancing between the load estimations in the subdomains, and one that considers only the load estimations of the interfaces. Algorithm 1 describes this selection procedure. Notice that, if both axes are equally balanced, both are selected and kept for further consideration. In the example of Fig. 2, for a threshold of $t = 0.20$, none of the axes would be discarded in the first test, but for a threshold of $t = 0.21$, for example, the Y axis would be selected. In this work, the threshold is set as $t = 0.50$, based on empirical tests and observations.

Algorithm 1 Axis selection.

```
Input: Subdomains $i$ and $j$ for $\alpha$ and $\beta$-axes, load estimations for each
    subdomain and each interface, threshold $t$.
Output: Axis or axes selected.
    ▷ Calculation of the balance factors
    $d_\alpha \leftarrow |(LE_i)_\alpha - (LE_j)_\alpha|$
    $d_\beta \leftarrow |(LE_i)_\beta - (LE_j)_\beta|$
    $(b_{ij})_\alpha \leftarrow d_\alpha / \max\{(LE_i)_\alpha, (LE_j)_\alpha\}$
    $(b_{ij})_\beta \leftarrow d_\beta / \max\{(LE_i)_\beta, (LE_j)_\beta\}$
    $(b_{ij})_{\alpha\beta} \leftarrow (LE_{ij})_\alpha / (LE_{ij})_\beta$
    $(b_{ij})_{\beta\alpha} \leftarrow (LE_{ij})_\beta / (LE_{ij})_\alpha$
    ▷ Tests using balance factors and threshold
    if $(b_{ij})_\alpha \le t$ and $(b_{ij})_{\alpha\beta} \le t$ then return $\alpha$
    if $(b_{ij})_\beta \le t$ and $(b_{ij})_{\beta\alpha} \le t$ then return $\beta$
    ▷ Tests using the balance between the subdomains
    if $d_\alpha < d_\beta$ then return $\alpha$
    if $d_\beta < d_\alpha$ then return $\beta$
    ▷ Tests using the interfaces
    if $(LE_{ij})_\alpha < (LE_{ij})_\beta$ then return $\alpha$
    if $(LE_{ij})_\beta < (LE_{ij})_\alpha$ then return $\beta$
    ▷ Return of the equally balanced axes
    return $\alpha$ and $\beta$
```

In two dimensions, the axis selection algorithm is performed only once, with the X and the Y axes. In three dimensions, it is performed twice, in the first pass, with the X and the Y axes, and in the second pass, with the axis selected in the first pass (notice that, in case both axes are selected, either X or Y can be used) and the Z axis.

After the tests, two, one or none of the axes is discarded. Thus, one, two or three axes are kept, but only one must be chosen. If two or three axes are kept, the chosen axis is the next available after the one that divided the parent node in the BSP tree structure. For example, if the parent node is decomposed on the X axis, and the Y axis is among the ones kept, it will be chosen; otherwise, the Z axis will be chosen. If the parent node does not exist, i.e., the decomposition is being performed for the root of the BSP, the first available axis is the chosen one.

3.2.3 *Parallel domain decomposition*

The decomposition procedure described in the previous section is performed until the number of subdomains is equal to the number of processes, in parallel, and in such a way that each process builds only the branch of the BSP necessary to reach its own subdomain. This can be done using the identifier of the process and the number of total processes, or the identifier of the thread and the number of threads in a process.

Figure 3 shows a global view of the BSP built on a domain (thick line) and the subdomains of each process (thin line). The parallel architecture in the example has eight processes. Figure 4 shows the BSP built in each process for the same example. Figure 5 shows the decomposition given by a BSP on the didactic example of a circle with uniform discretization, for eight subdomains. It can be seen that the two leftmost and the two rightmost subdomains have larger width than the four innermost subdomains, to make them have approximately the same load.

3.2.4 *Proportionality factor*

To enable the use of a number of processes different than a power of two, a proportionality factor is used during the construction of the BSP. If this proportionality factor is not used, when the number of processes is not a power of two, some of the generated subdomains will have approximately only half of the load estimations of the other subdomains. This could cause imbalances on the work distribution and, occasionally, lead to a poor speed-up.

With the proportionality factor, whenever the load estimations of two subdomains are compared to each other, their values are multiplied by a scalar that specifies how heavy one subdomain must be with respect to the other. This implies that, when a BSP cell is subdivided with a factor of $m : M$, one of its children will generate m subdomains while the other will generate M subdomains. The proportionality factor for a total of N subdomains is $\lfloor N/2 \rfloor : \lceil N/2 \rceil$, for each child to generate approximately half of the final subdomains.

For example, if the number of processes is five (Fig. 6), the proportionality factor used for the root of the BSP is $2 : 3$. That factor specifies that, when the load estimations for the root's children are compared, the load of the lighter child is multiplied by three and the load of the heavier child is multiplied by two. During the decomposition on an axis, if these scaled load estimations are equal, the recursive

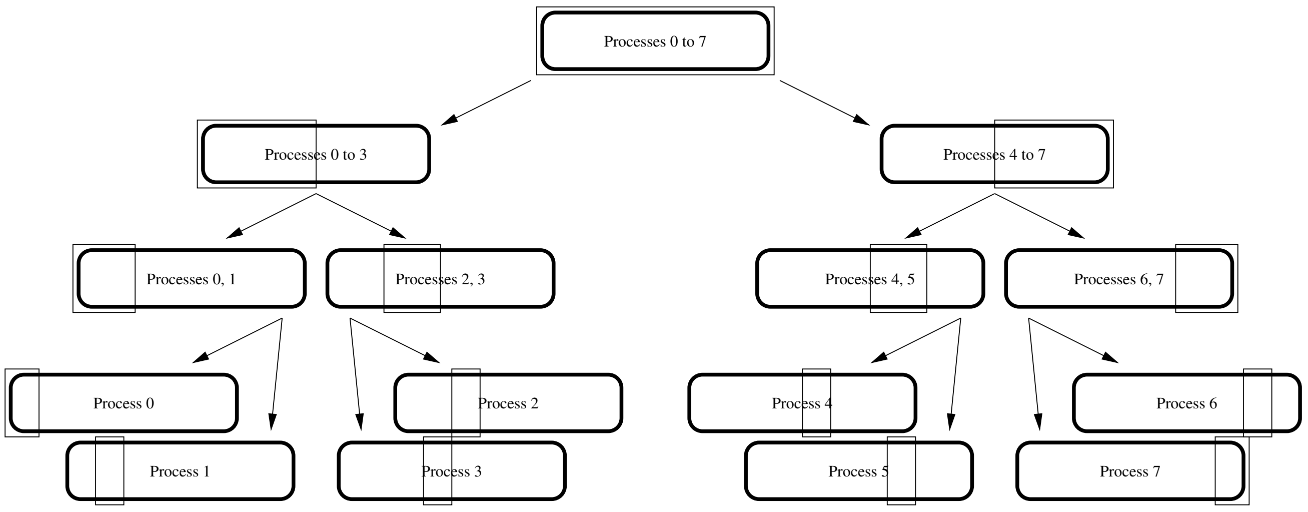

Fig. 3 Global view of the BSP

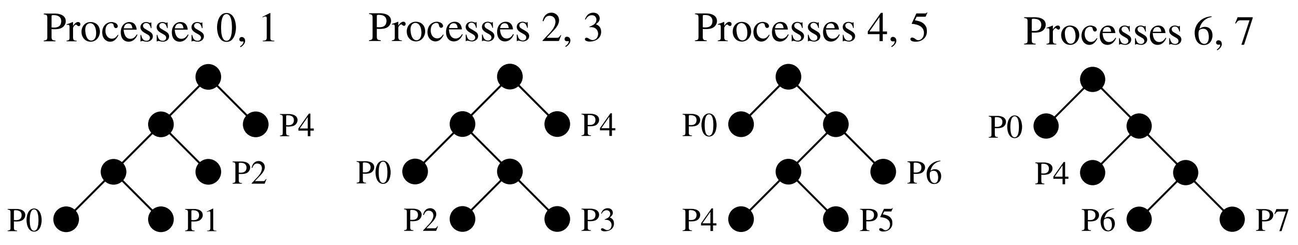

Fig. 4 Local BSP as seen in each process. A label indicates the process responsible for a branch

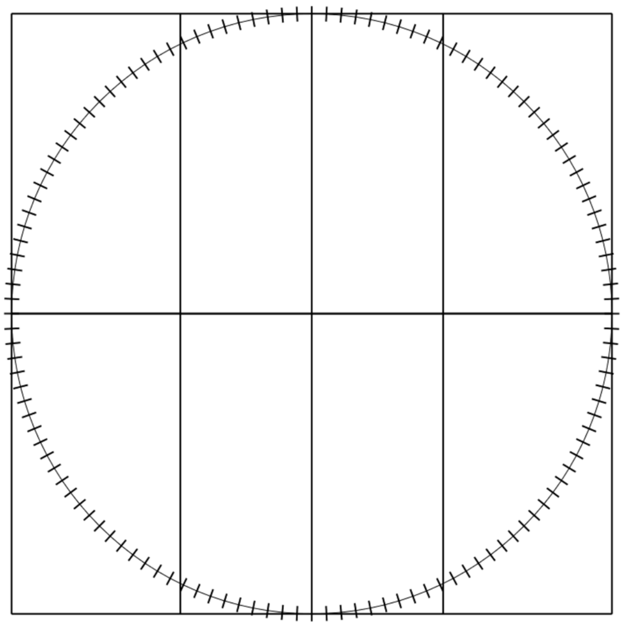

Fig. 5 Decomposition performed by a BSP

procedure stops; otherwise, the originally heavier child is taken and the decomposition plane is moved towards its direction. In the end, the decomposition plane's position is the one that minimizes the absolute difference of the scaled load estimations. That same idea used for deciding the best decomposition axis.

In the example of Fig. 6, the heavier child of the BSP root is subdivided with a proportionality factor of 1:2, while the lighter child has a proportionality factor of 1:1. Finally, the heavier child of the heavier child of the BSP root is further subdivided, with a proportionality factor of 1:1. Using the proportionality factor results in five subdomains with approximately identical load estimations.

3.3 Mesh generation

3.3.1 Mesh generation on a subdomain

Since the algorithm used in each subdomain is based on an advancing front technique (AFT), an initial front is needed in all subdomains. However, when the BSP is created, it is

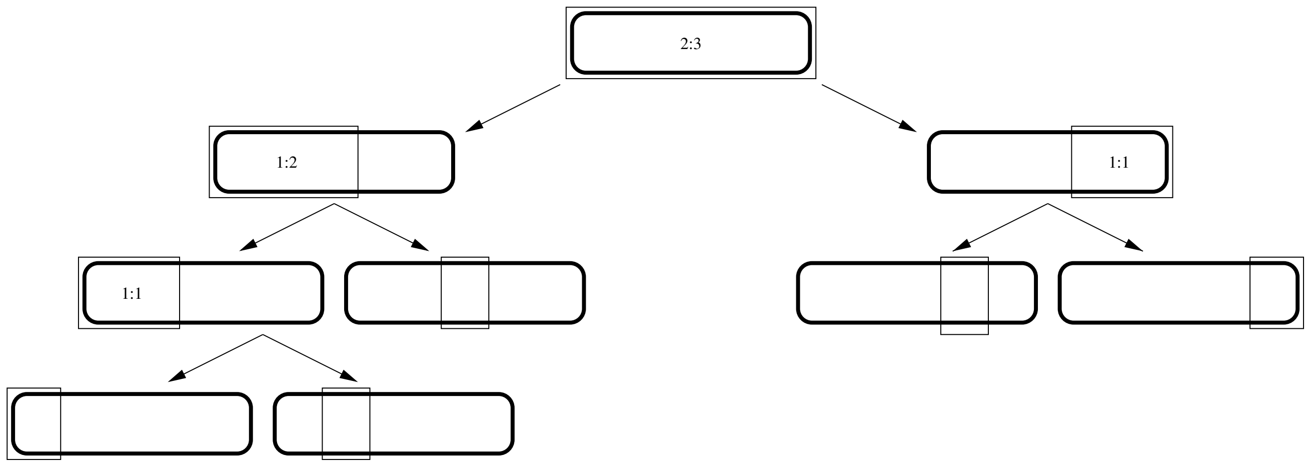

Fig. 6 Example of a BSP tree with five subdomains. The proportionality factors are indicated in each internal node of the BSP

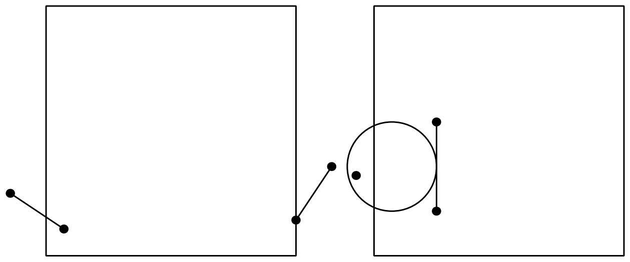

Fig. 7 Advancing front modifications (2D case): segments crossing the bounding box (*left*) and search region crossing the bounding box (*right*)

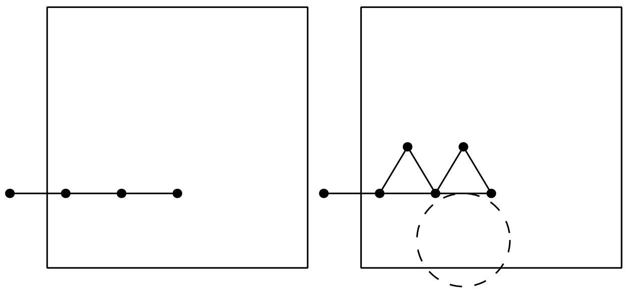

Fig. 8 Crack and conditions applied to the AFT

possible that some subdomains remain completely internal to the domain. In those cases, some auxiliary faces are generated near the center of the subdomain, with sizes that respect the sizing function given as input to the AFT. That can be done because the auxiliary front lies near the center of a subdomain that does not cross the initial front and, therefore, the auxiliary front is far from the boundary of the domain.

Each subdomain, defined by a part of the front and a bounding box, generates an independent sub-mesh using a serial AFT developed by some of the coauthors of this work [1–3]. To ensure that no element is generated outside the limits of a subdomain, the following conditions were added to that serial AFT:

1. A face crossing the bounding box of the subdomain is not advanced (Fig. 7, left).
2. A face that is strictly inside the bounding box of the subdomain is not advanced if any valid well-shaped element formed with it would cross the bounding box. In other words, consider the search region that is defined for placing a new vertex, which, when connected to the given face, would form a valid, well-shaped element. Then, if that region crosses the bounding box, the face is not advanced (Fig. 7, right).

If the second condition were modified just to test whether or not the ideal vertex for a given face is outside the bounding box, there would be too little empty space left for generating the interface mesh *a posteriori* (see discussion in Sect. 3.3.2). Therefore, the whole search region is tested.

No special treatment is necessary for crack faces, as they follow the same conditions. This is an advantage of this technique, since crack faces take no extra processing time, and are treated as any other face on the boundary. Figure 8 depicts both conditions, for the case of 2D crack segments: according to the first condition, the leftmost segments cannot advance, because they cross the subdomain's bounding box; and, according to the second condition, the segments facing down also cannot advance, because their geometric search regions cross the subdomain's bounding box.

In cracks, due to the existence of vertices with identical coordinates, the advancing front algorithm must know how

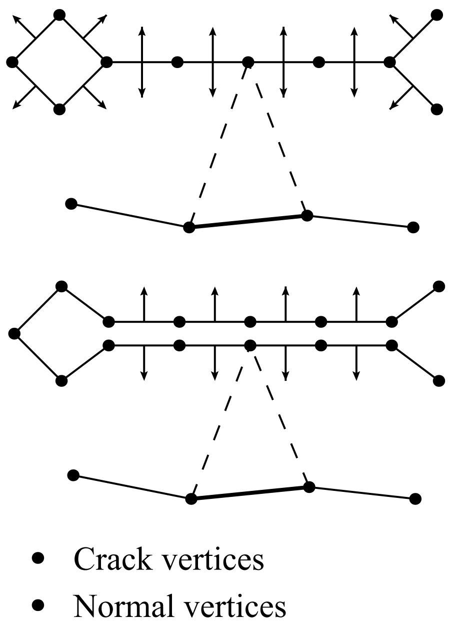

Fig. 9 Geometrical (*top*) and topological (*bottom*) views of a base segment advancing to a 2D crack vertex. The distance between crack vertices in the topological view is only schematic

to distinguish between them and choose the appropriate vertex for the generation of a new element, avoiding topological inconsistencies (Fig. 9). For that purpose, the algorithm uses the faces of the input front that are adjacent to those vertices. When two candidate vertices on the crack are selected to generate a new element, the one on the same side of the base face, with respect to the crack, is chosen. The normals of the faces adjacent to the vertices are used to perform that test.

Each process advances the front as much as possible, respecting the two previous conditions. For example, the top left picture of Fig. 10 shows a 2D initial front of a subdomain. The same figure also shows its updated front, i.e., the remaining front after the sub-mesh was generated, along with its sub-mesh. An improvement is applied to the sub-mesh, as a combination of Laplacian smoothing, face swapping and edge swapping procedures. More details about those algorithms can be found in [1–3]. Notice that the updated front cannot change when smoothing or any other optimization technique is performed, since some of its adjacent elements have not been generated yet.

3.3.2 Mesh generation on the interfaces

The meshes on the interfaces of the subdomains are generated as the BSP tree is ascended. After generating the mesh on a leaf of the BSP, its parent cell is determined and the identifiers of the processes that own each of its two children are retrieved. The process with the smaller identifier will be responsible for the parent cell. Then, through local

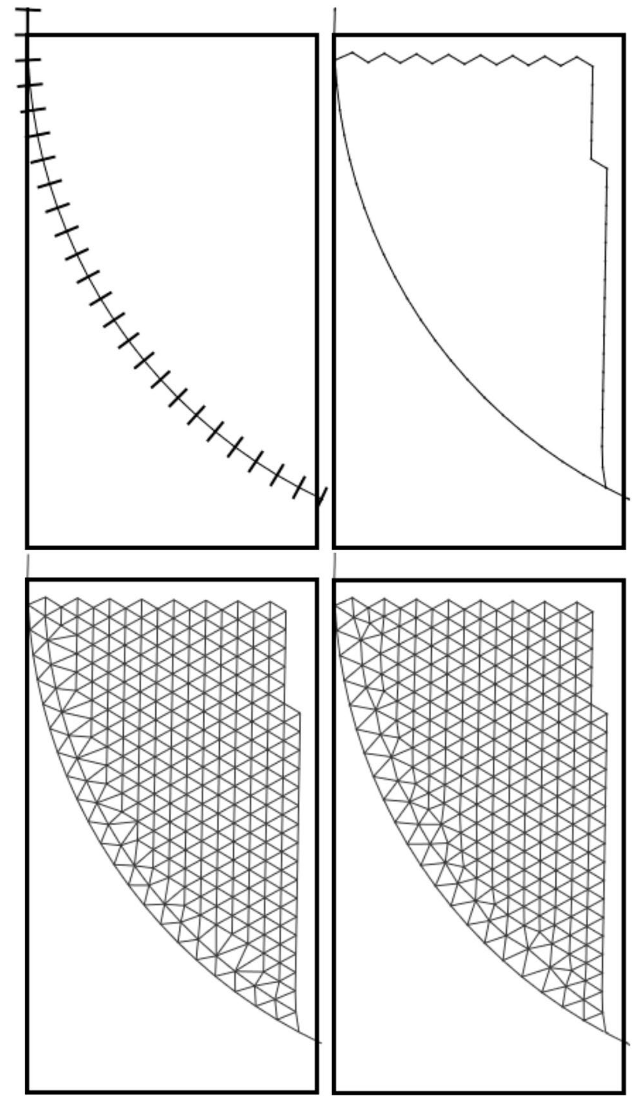

Fig. 10 Advancing front on a subdomain. *Top row*, from *left to right* initial front and bounding box of the subdomain, and updated front; *bottom row*, from *left to right* mesh before improvement, and mesh after improvement

synchronization, the process combines the updated fronts of the two children. Figure 11 shows the sub-meshes for the circle's example.

The updated fronts of the two neighboring subdomains, along with one layer of elements adjacent to those fronts, are gathered, and they are the starting point of the AFT applied to the interfacing region. That layer of elements is necessary for the mesh improvement to be performed on the updated fronts of the neighboring subdomains, which were not allowed to change during the improvement of the two sub-meshes. The AFT and the mesh improvement procedures applied to the interface are the same ones applied to the subdomains. Figure 12 shows the interface between the two bottom left subdomains of the circle's example.

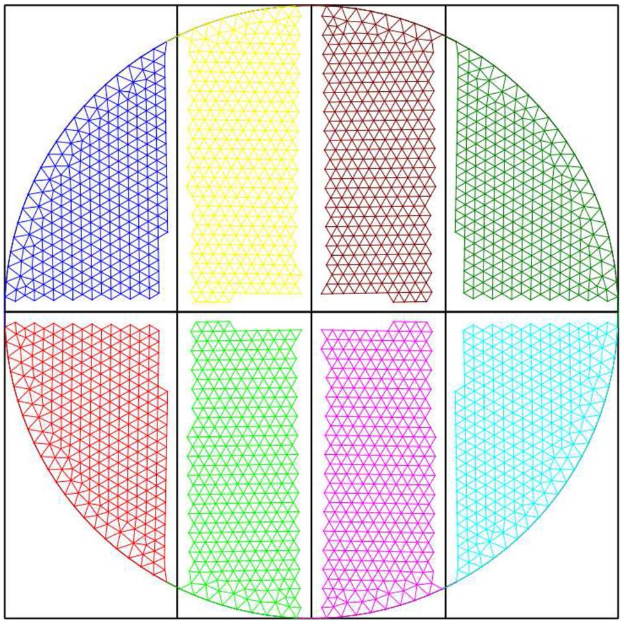

Fig. 11 Sub-meshes in the BSP leaves

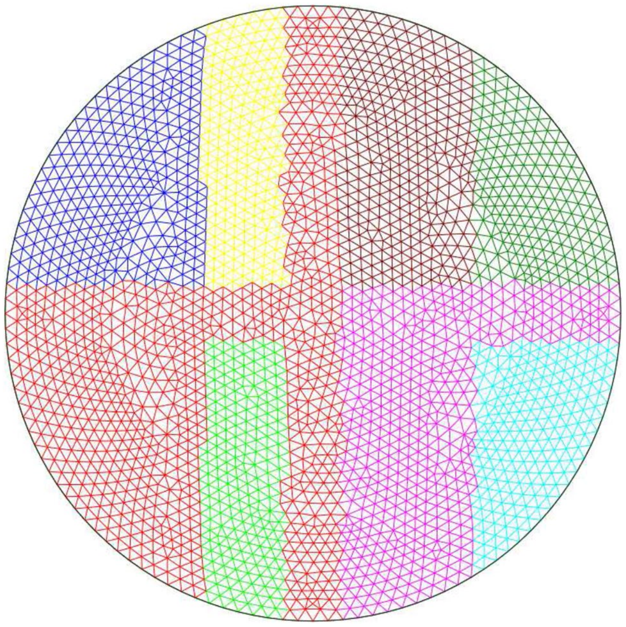

Fig. 13 Final mesh

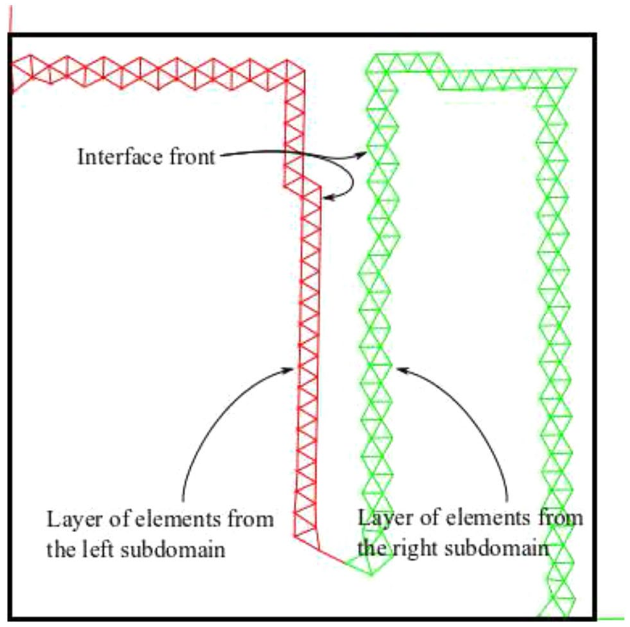
Interface front
Layer of elements from the left subdomain
Layer of elements from the right subdomain

Fig. 12 Interface region where a mesh will be generated with one layer of elements from each BSP child (*bottom*)

This procedure is repeated until the root of the BSP is reached, in which case the process with identifier 0, which can be seen as the master process, finalizes the mesh's generation. In the end, the generated mesh remains scattered among all the processes. Figure 13 shows the final mesh for the example of a uniform disk. Processes 0–7 are represented by the colors light red, light green, blue, yellow, magenta, cyan, dark red and dark green, respectively. Figure 14 depicts a global view of the mesh generation on the interfaces of the subdomains.

4 Results

The parallel technique was developed in C++ with MPI for interprocess communication. It was executed in the cluster computer Stampede, maintained by the Texas Advanced Computing Center, and it used the GNU Compiler Collection version 4.7.1 and the MPI implementation MVAPICH version 1.9a2.

The tests presented in this section consist in generating meshes for four models: A prismatic beam with an extremely refined boundary mesh; a rotor; a cracked gear; and a gear with no crack. The first test is a didactic test devised to assess whether our technique would be able to deliver a linear or even super-linear speed-up as expected for such models. All the other three models are real engineering models. The third and fourth models are identical, except for the presence of a crack in the third model and for the higher boundary mesh refinement in the fourth model. The existence of a crack in the gear forces the generation of a high number of elements in the 3D mesh. To have comparable number of elements in the third and fourth models the input boundary mesh for the gear without crack was highly refined.

The models can be seen in Fig. 15, and their decomposition with 16 processes can be seen in Fig 16. Figure 17

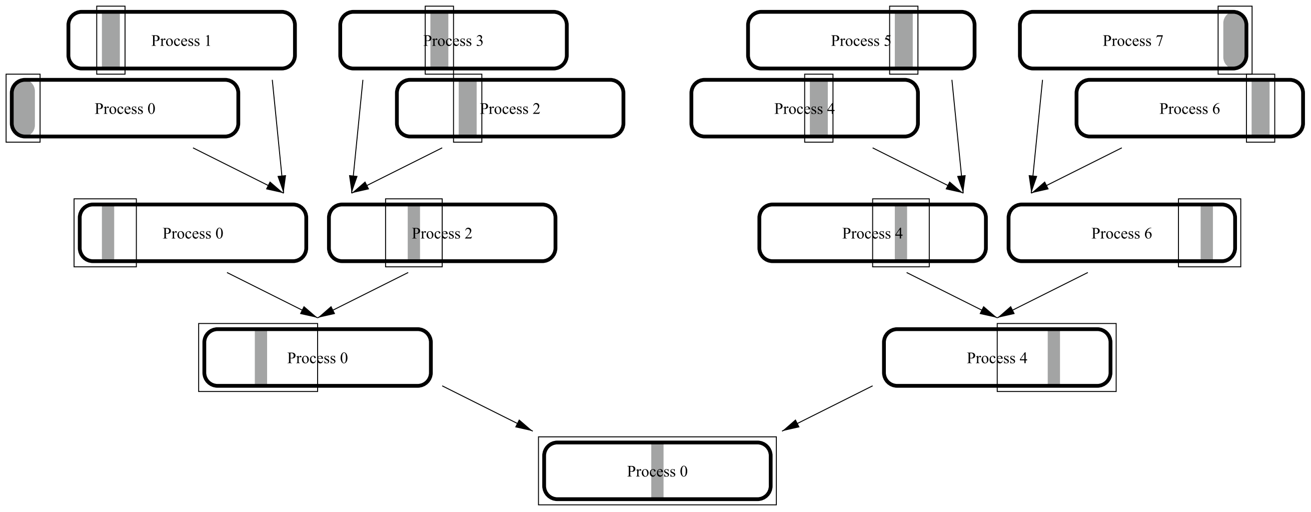

Fig. 14 Global view of the mesh generation on the interfaces

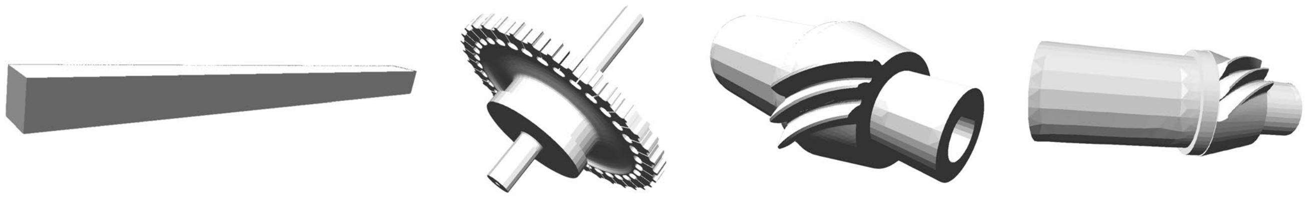

Fig. 15 Test models: beam, rotor, cracked gear, and gear

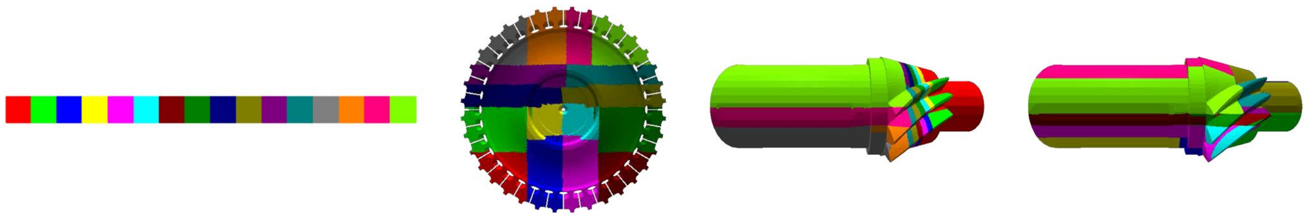

Fig. 16 Frontal view of the decomposition of the test models: beam, rotor, cracked gear, and gear

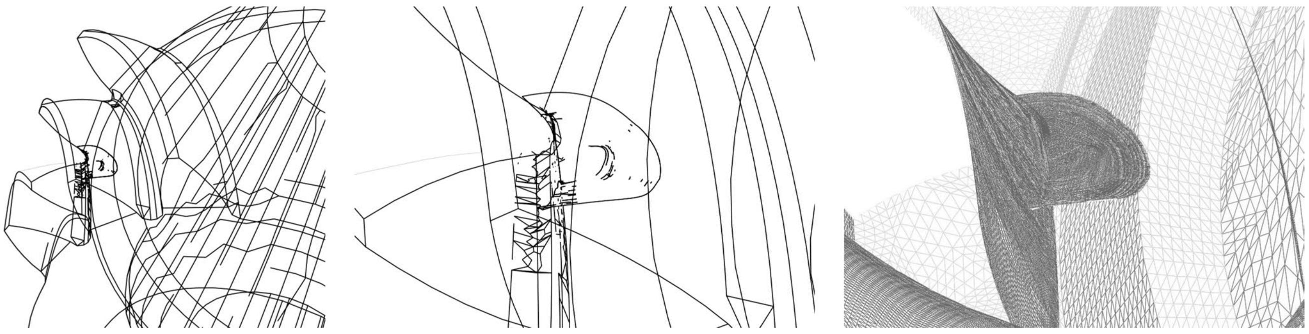

Fig. 17 Zoomed view of the crack in the cracked gear model

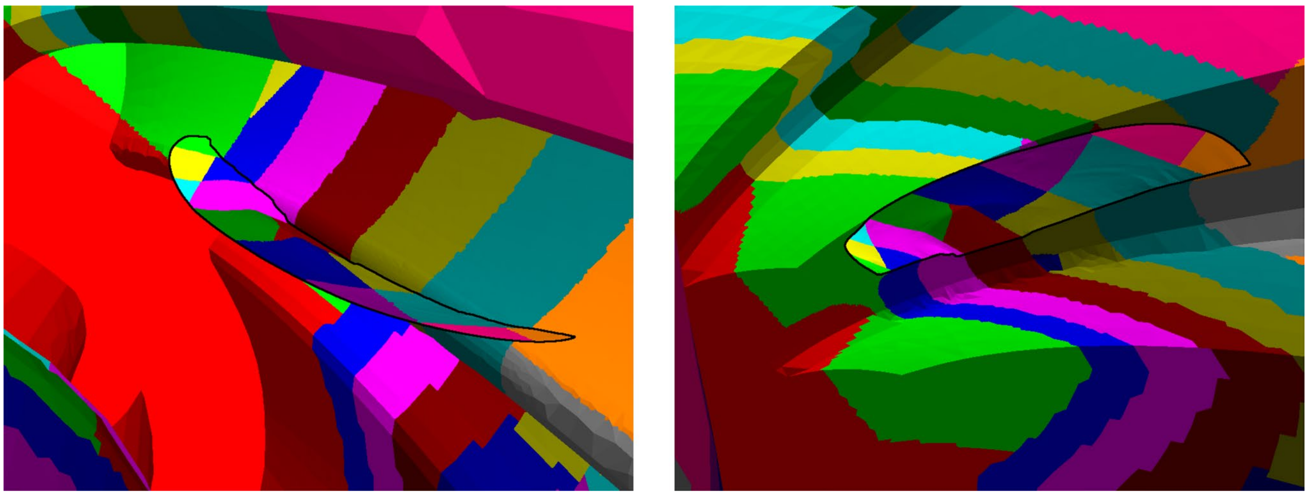

Fig. 18 Decomposition of the crack in the cracked gear model, viewed from above the crack (*top*) and from under it (*bottom*)

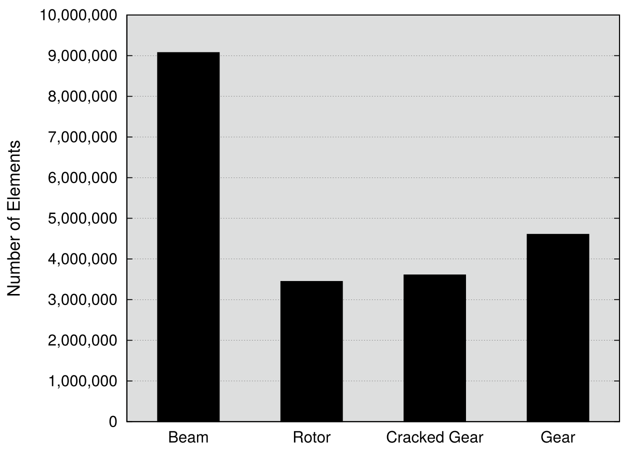

Fig. 19 Mesh sizes for each model

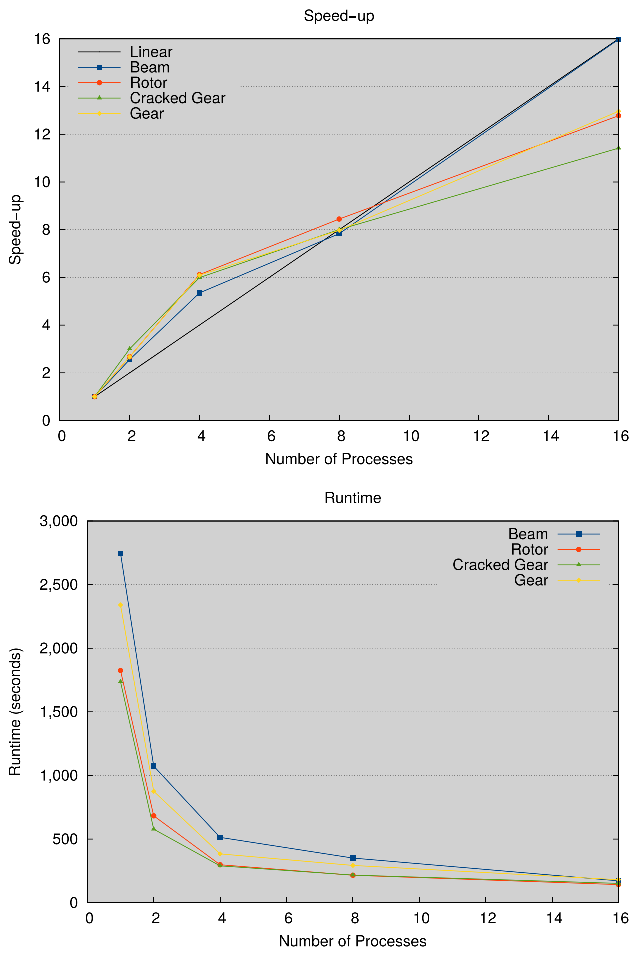

Fig. 20 Runtime and speed-up of each model

shows the crack in the cracked gear model and Fig. 18 shows how it is affected by the decomposition. Figure 19 shows the size of the meshes generated by the serial implementation.

4.1 Runtime and speed-up

Figure 20 shows the runtime and the speed-up obtained with the parallel implementation. Figure 20 (top) shows that the runtime of the serial implementation varies with the size of the generated mesh. The meshes for the rotor and the cracked gear models had approximately the same size and, thus, they took approximately the same time to be generated. The mesh for the gear was a little larger because its boundary is more refined, and, therefore, its generation took longer. The mesh generated for the beam model was the largest mesh, and, therefore, took the longest time to compute. However, notice that, due to the simplicity of the geometry, the difference between its runtime and the one for the gear model was not very large, even though the mesh of the beam model has approximately twice the size of the mesh of the gear model.

Figure 20 (bottom) shows that the parallel implementation, in general, delivered a good speed-up in all the models for up to 16 processes. Using 2 and 4 processes, a super-linear speed-up was delivered. Using 8 processes, the speed-up obtained was linear and, using 16 processes, the speed-up decreased, although the worst speed-up was of almost 12, for the cracked gear model, which is still a very good result. This drop was caused by some imbalance due to the existence of the crack. The speed-up for the gear model without crack, in comparison, was the same as for the rotor model. The case of the beam model maintained the linear speed-up, which attests that this implementation is fine tuned.

4.2 Runtime in detail

Figure 21 shows a profile of the master process that indicates the time spent in each activity of the parallel mesh generation algorithm: initialization; boundary building, which creates the data structures of the input front; generation of the load estimation octree, which generates the octree and classifies its cells (detailed in Sect. 3.1); Computation of the load, which counts the number of full cells (also in Sect. 3.1); Generation of the BSP data structure (Sect. 3.2); Decomposition of the front, which distributes the input front's faces among the subdomains; Generation of the meshes in the subdomains (Sect. 3.3.1); Communication, which synchronizes the master process with its neighboring processes; Merging of parts of the meshes generated by the processes; Generation of the interface meshes (Sect. 3.3.2); and Overhead, which accounts for any wasted time not associated with any of the activities. Figure 22 is analogous to Fig. 21, but expresses the time spent in a given activity as a percentage of the total time.

The time to generate and classify the octree is constant, it does not depend on the number of processes. The dominant time is the mesh generation procedure. As the number of subdomains increases, the number of interfaces also increases, but only a minor growth of time to mesh them is observed, especially from 4 to 16 processes.

Figure 22 shows that, as the number of processes increases, the total runtime decreases and, therefore, constant or nearly-constant times become proportionally high. For example, with 16 processes, the octree generation and classification step takes 15–25 % of the runtime, and the generation of the interface mesh ranges from 15 to 45 % of the total runtime.

In a simple best-case scenario, if only the generation and classification of the octree were parallelized in an ideal manner, the overall speed-up for 16 processes would be almost linear, as shown in Table 2. This indicates that the method would scale well for a higher number of processors.

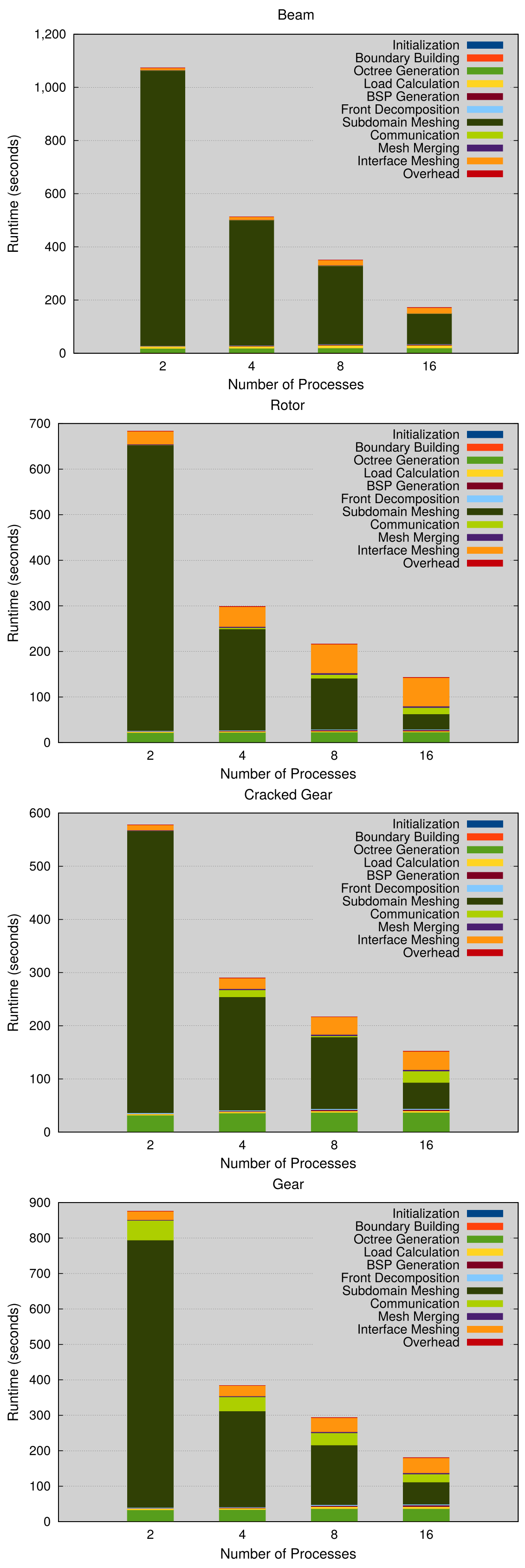

Fig. 21 Detailed runtime

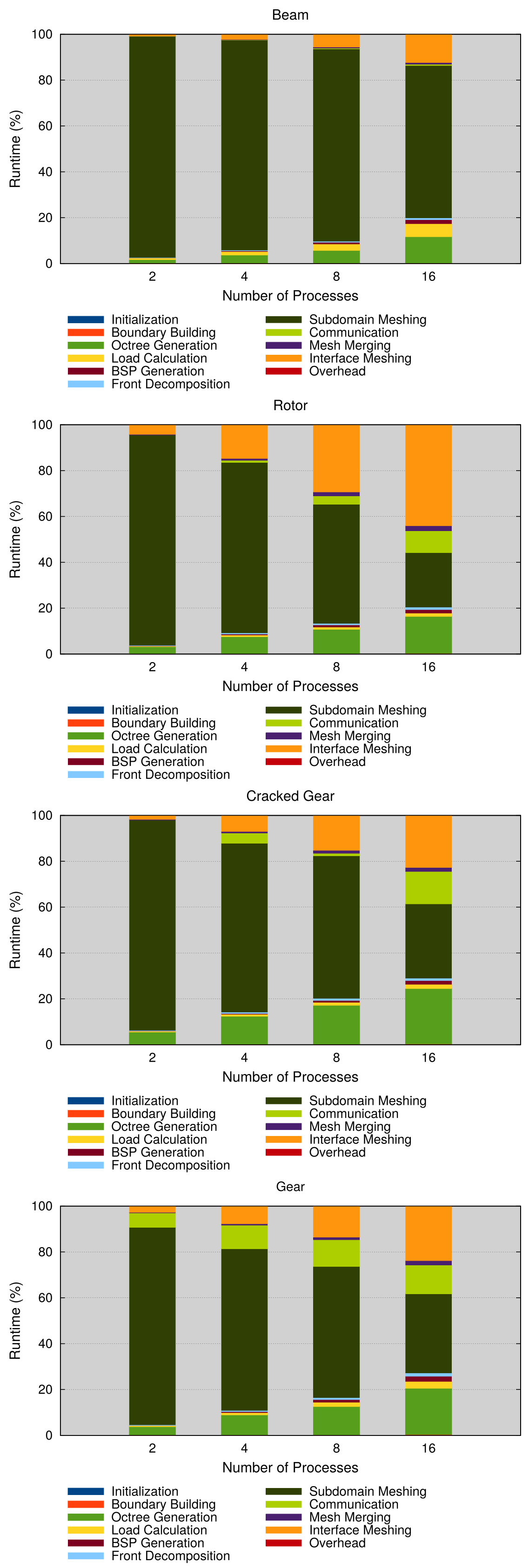

Fig. 22 Detailed runtime in percentage

Table 2 Speed-up for 16 processes with an ideally parallelized octree.

Model	Speed-up
Beam	17.9
Rotor	15.0
Cracked gear	14.7
Gear	15.9

4.3 Load balancing

Figure 23 shows the runtimes of subdomains and interfaces in each process for the four models when 16 processes were used. The first task of a process is executed in the subdomain assigned to it, and its subsequent tasks are directed to the mesh generation in the interfaces. However, the task of generating mesh in the interfaces is assigned to just a subset of the processes.

In the beam model, the amount of work in each process was well balanced due to its regular decomposition. The rotor model has well-balanced subdomains and well-balanced interfaces in each level of the BSP tree. However, from level to level, there was some imbalance, which caused a small speed-up loss. The cracked gear model had the worst load balance, because the load estimations for the subdomains near the crack were poor. The removal of the crack improved the distribution of the load among the processes considerably. In both gear models, the loads assigned to meshing most interfaces were small.

Load balance is closely correlated to runtime. In fact, when the number of elements generated in each process is well-balanced, the runtime in each process is also well-balanced. That can be seen in the charts presented in Fig. 23 (runtimes) and Fig. 24 (number of elements).

4.4 Load estimation

The load estimation given by the octree in Section 3.1, for either a subdomain or an interface, is a dimensionless parameter that, when divided by the sum of all the load estimation parameters, indicates the percentage of the total number of elements of the mesh that will be generated in that subdomain.

That percentage can be applied to estimate, roughly, the number of vertices or the runtime associated with the mesh generation in a subdomain. However, since the exact percentage could only be computed after the whole mesh is generated, what is actually computed is an estimated percentage. Figure 25 shows the estimated number of elements and the number of elements generated in each subdomain when 16 processes were used.

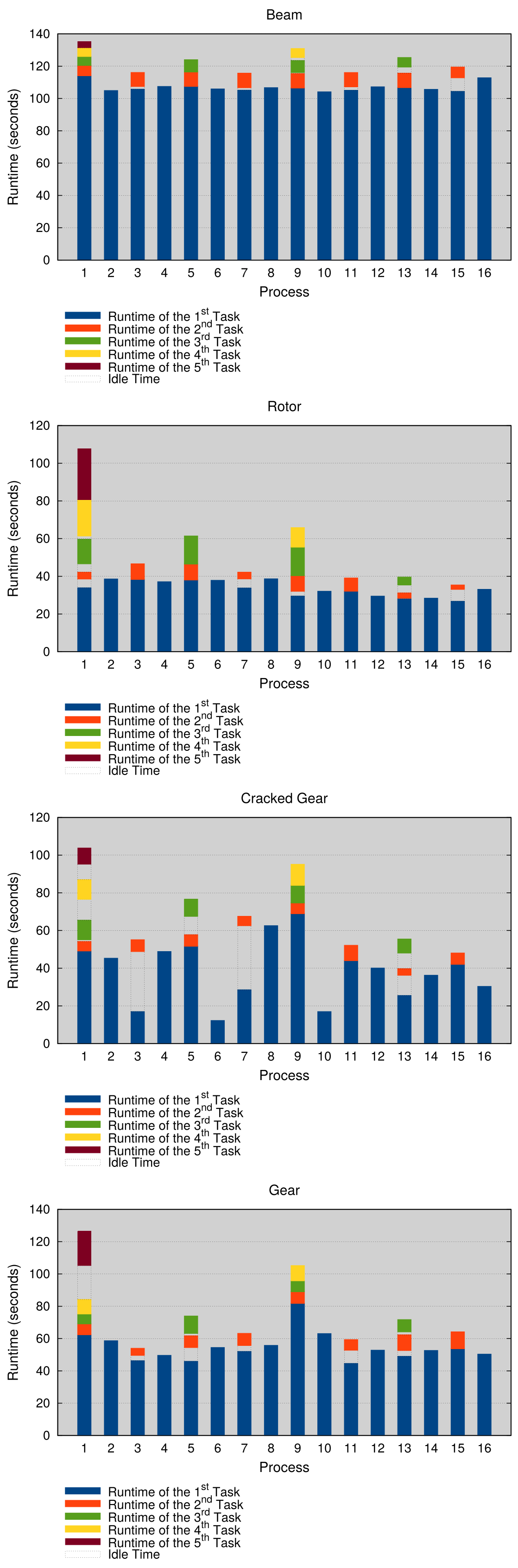

Fig. 23 Runtime of the mesh generation tasks in each process

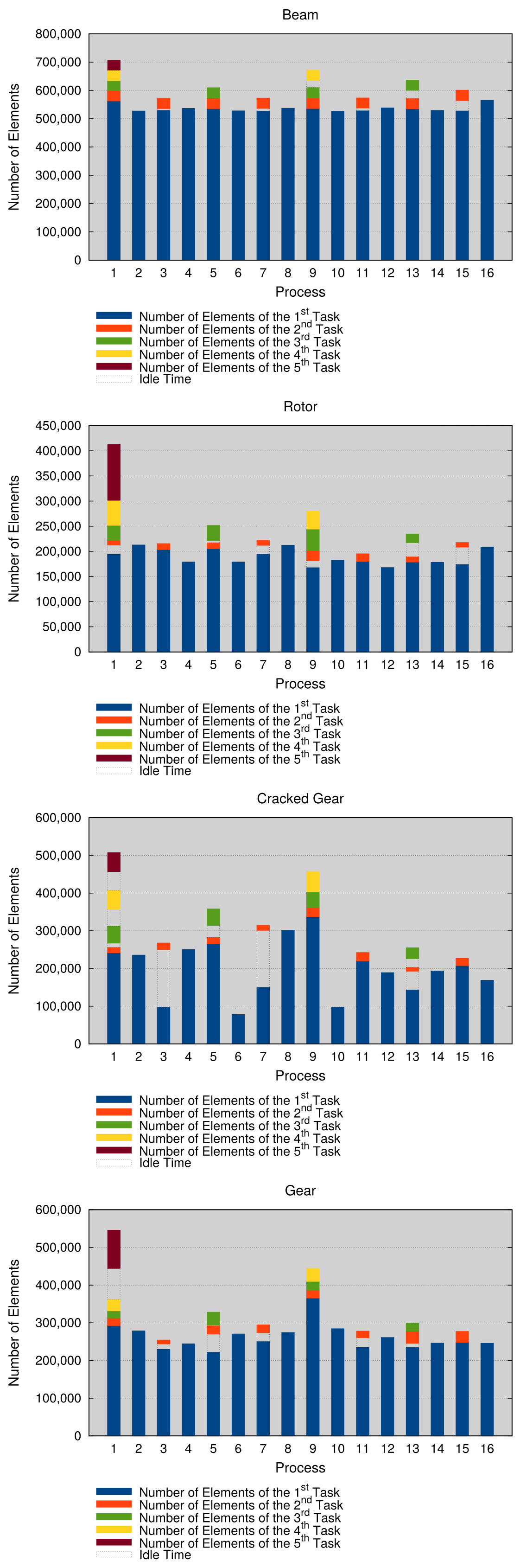

Fig. 24 Number of elements of the mesh generation tasks in each process

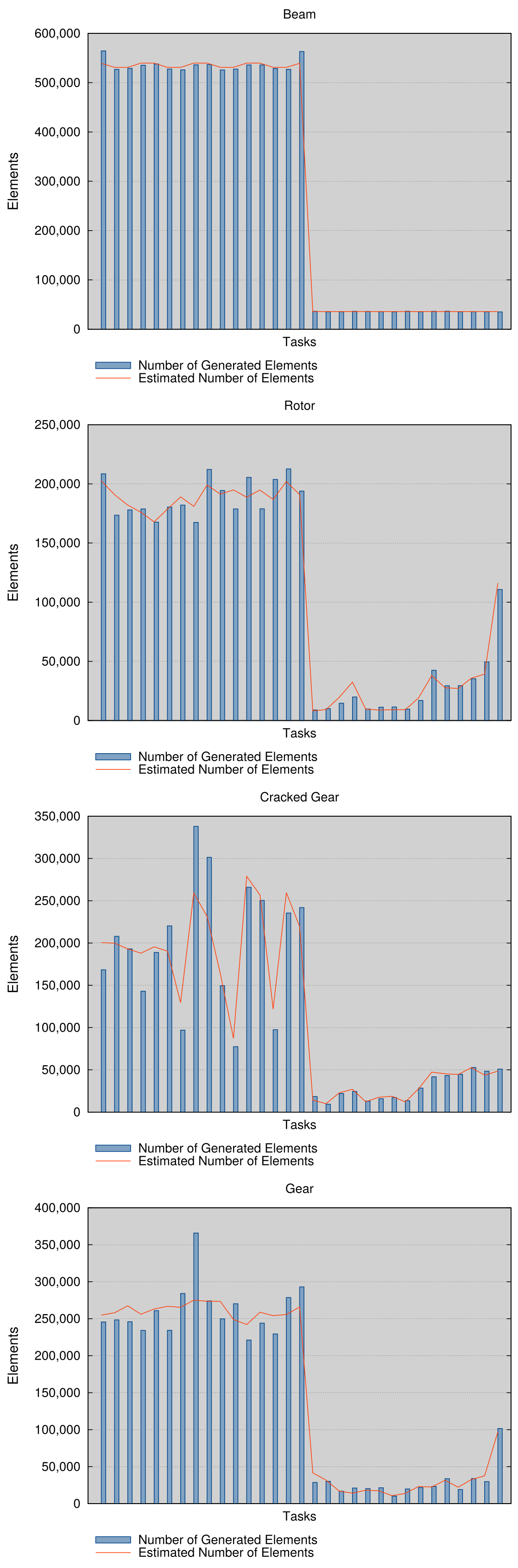

Fig. 25 Estimation of the number of elements for the 16 subdomains and for the 15 interfaces

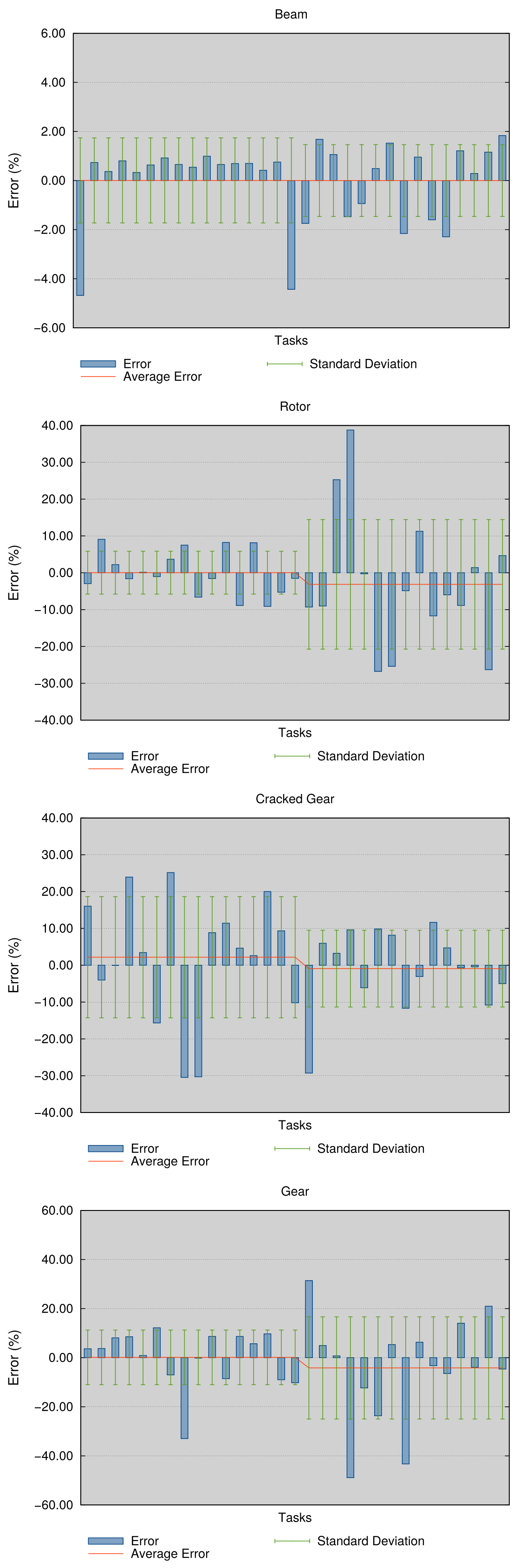

Fig. 26 Error in the estimation of the number of elements for the 16 subdomains and for the 15 interfaces

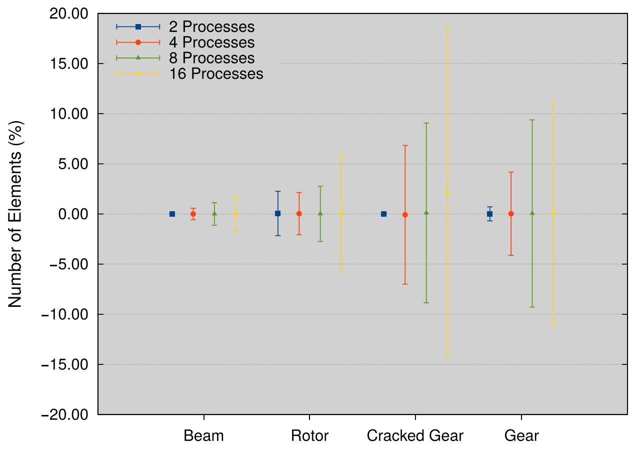

Fig. 27 Average error in estimation of the number of elements and standard deviation of the error

The didactic test with a beam model showed good estimation of the number of elements, as can be seen in the first chart. The rotor model also showed some uniformity. The cracked gear model showed that the estimation was not very good, because of the crack. Its removal, in the gear model, led to a better uniformity, except for one subdomain. The crack itself is not the problem, but it would possibly induce the generation of many small elements in the interior of the model, which might cause difficulty in the estimation of the load by the octree. However, although the load estimation for the example with crack is not as good as those for the cases without crack, as shown in Fig. 25, the estimation is still reasonable.

The error of the estimation of the number of elements can be calculated as the relative error between the estimation and the actual number of elements, as depicted in Fig. 26. A positive error means that the number of elements was overestimated, while a negative error means that the number of elements was underestimated. Figure 26 also shows the average error and the standard deviation of the error. If the standard deviation approaches zero, most of the individual errors approach the average error. If the average error also approaches zero, most of the individual errors approach zero, which indicates that the estimated number of elements approaches the actual number of elements, and, therefore, the load is well estimated. As expected, the cracked gear model showed the worst error of the four models, when 16 processes were used.

Figure 27 shows the average error and the standard deviation of the error for the subdomains for all the runs of the parallel implementation. It can be seen that, as the number of processes increases, thus increasing the number of decomposition planes, the error increases. However, except for the cracked gear model, the average error remained small, only the standard deviation was increased. Notice that this error, for most cases, was not larger than 10 %.

4.5 Mesh quality

The main purpose of the parallel mesh generator is not to generate a better mesh, but to generate more rapidly a mesh whose quality is not too far from the mesh that would be generated by the serial mesh generator. In this work, the quality of an element was calculated as the ratio between the radii of its inscribed and its circumscribed spheres, multiplied by 3. This ratio varies from 0 to 1, and small values (less than 0.1) indicate elements of low quality, while large values (larger than 0.7) indicate elements of good quality. The value of 1 indicates that the tetrahedron is equilateral.

Figure 28 shows the percentage of the number of elements in each range, varying from 0 to 1, with intervals of size 0.1, for all the generated meshes, including the serially generated meshes. It can be seen that, as the number of processes increases and, therefore, the number of decomposing planes also increases, the quality of the meshes worsens somewhat. However, this worsening is mostly caused by the loss of quality of a few elements from the high quality range (greater than 0.7) going down to the average quality range (between 0.4 and 0.7). This can be better seen in Fig. 29, which depicts the difference, in each range, between the quality, in percentage, of a mesh generated in parallel and the one generated serially. A positive value means that the parallel technique with that number of processes generated more elements in a range than the serial technique. A negative value means the opposite. It can also be seen that the largest difference, for the rotor model with 16 processes, was less than 3 %, in absolute value.

To better measure how different a mesh generated in parallel is from a mesh generated serially, the absolute values of the differences show in Fig. 29 for each range were summed, and the results were depicted in Fig. 30, i.e., the plots show the total difference for each mesh generated in parallel. The mesh generated with 16 processes for the rotor model presented the highest difference, which is approximately 10 %. For the other models, the total difference reached, at most, 5 %. Notice also that even though the speed-up and the load estimation for the cracked gear model could be better, the quality of the meshes generated in parallel are close to the quality of the mesh serially generated. This means that the quality of the mesh is independent of the speed-up and of the load estimation.

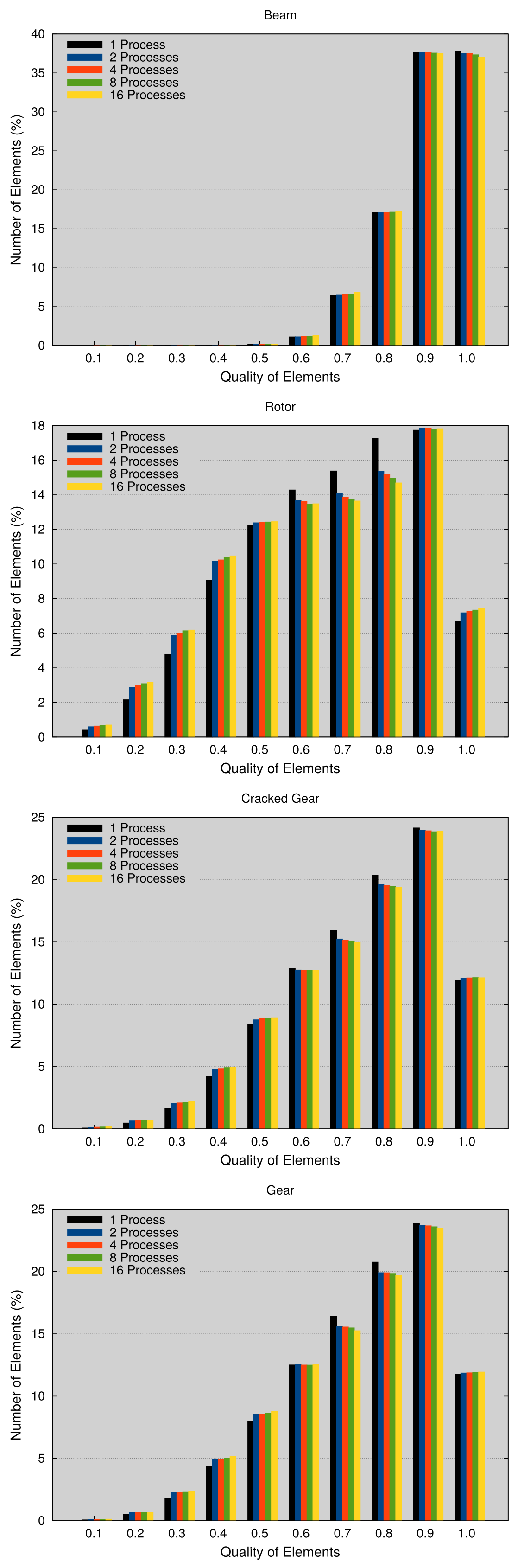

Fig. 28 Percentage of the quality of the meshes

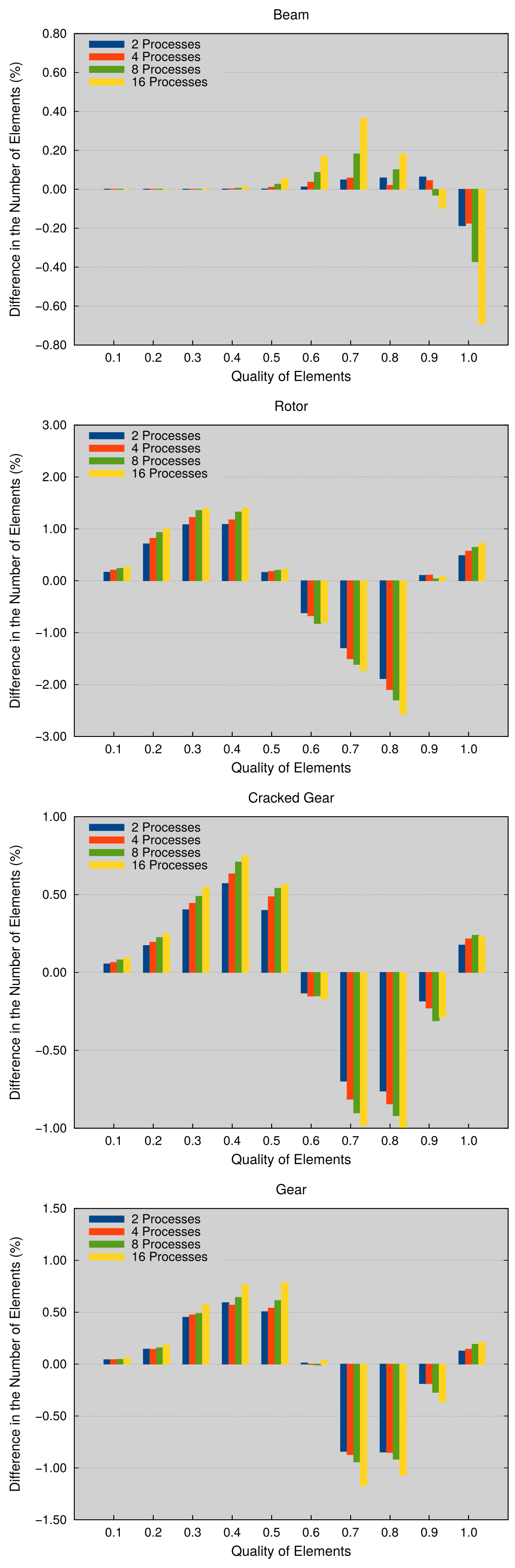

Fig. 29 Difference in the percentage of the quality of the meshes (parallel vs. serial)

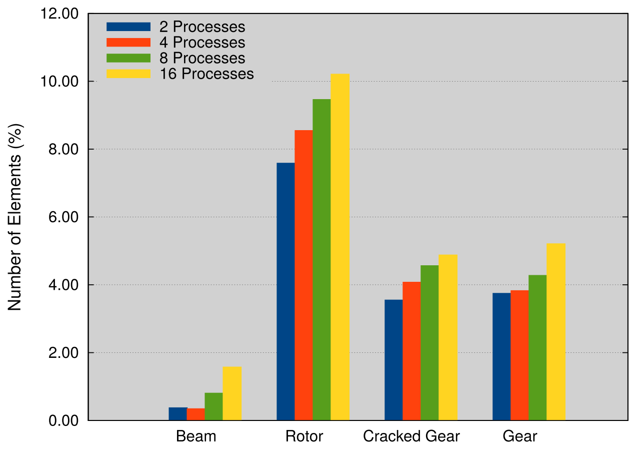

Fig. 30 Total difference in the quality of the elements (parallel vs. serial)

5 Conclusions

This work presented a technique for generating meshes in parallel for shared, distributed or hybrid-memory machines. The technique does not require any modification to deal with models with cracks and models without cracks. The technique uses an axis-aligned binary spatial partition (BSP) tree to decompose the domain according to the number of processes or threads. That decomposition is based on a fine octree, that works as a density function for the interior of the domain, so that all the subdomains have approximately the same workload. Even though the binary behavior of the BSP tree suggests that the number of threads or processes be a power of two, a proportionality factor is used in this work. That factor ensures that, even if any other number of processes or threads is used, the generated subdomains have approximately the same workload, which is also another advantage of our technique.

The subdomains are meshed in parallel and the meshes interfacing them are generated as the tree structure is ascended, locally synchronizing only the two processes responsible for the two neighboring subdomains when necessary. The serial mesh generator used in this work is based on an advancing front procedure capable of dealing with cracks. Thus, because the continuous domain decomposition technique does not restrict the input data, the parallel technique also can deal with cracks. This parallel technique is generic enough to be used in two or three-dimensions.

The results show that the technique delivered a good speed-up in all the models for up to 16 processes. Regarding load balance, the main concern of this work was to balance the number of elements generated in each subdomain, and the results were very satisfactory. The load estimation was very reasonable, with the average error remaining small, for all models. Although the main purpose of a parallel mesh generator is not to generate a better mesh, but rather to generate more rapidly a mesh whose quality is not too far from the serially generated mesh, the quality of the meshes generated by the presented parallel technique was very good and very close to the serial version.

Furthermore, the results showed that the existence of the crack can lead to an imbalance, due to the existence of too many elements unaccounted by the load estimation technique. In all the other models, the load was well estimated, with errors below 10 %, leading to a good load balance. Even with the presence of some imbalance, the speed-up delivered by the implementation was good enough for its use, especially because the quality of most of the meshes generated in parallel was close to the quality of the serially generated one, within an error margin of 10 %.

Although results were shown only for distributed-memory computers, the technique does not impose restrictions on the type of memory, and it can be easily implemented using shared-memory constructions, such as lock mechanisms for one thread to wait for another on the generation of interfacing meshes. The shared-memory implementation will be addressed in the future.

The main concern regarding this technique is that it is heavily dependent on a good load estimation, and the efforts in the future will be directed towards that issue. One way to improve the load estimation is to modify it in such a way that the contribution of each octree cell vary with its depth in the tree structure. Also, it is known that the octree might not be the best choice for models that are not aligned with the global axes. Therefore, it must be built in a local coordinate system, a matter that will be addressed in the future. Furthermore, substituting the time spent in generating the load-estimation octree sequentially by the time spent for generating that octree in an ideal parallel situation, the speed-up drop is nearly eliminated, which indicates that the method would scale well for a higher number of processors. Finally, the technique can be further optimized by the generation of the interfacing meshes prior to the generation of the meshes in the subdomains, although this can create artifacts on the final mesh that can degrade its overall quality.

Acknowledgments The first author acknowledges the support from the Brazilian agency CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) through the fellowship granted to him (Process 2823-12-8). The third author acknowledges the support from the Brazilian agencies CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) through the Research Productivity Grant 307941/2013-2, CAPES through the grant BEX 6881-12-2 and FUNCAP (Fundação Cearense de Apoio ao Desenvolvimento Científico e Tecnológico) through the grant CI1-0080-00064.01.00/13. All the authors acknowledge the Texas Advanced Computing Center (TACC) (http://www.tacc.utexas.edu) at The University of Texas at Austin for providing high performance computing resources that have contributed to the research results reported within this paper.

References

1. Cavalcante-Neto JB, Wawrzynek PA, de Carvalho MTM, Martha LF, Ingraffea AR (2001) An algorithm for three-dimensional mesh generation for arbitrary regions with cracks. Eng Comput 17(1):75–91
2. de Oliveira Miranda AC, Cavalcante-Neto JB, Martha LF (1999) An algorithm for two-dimensional mesh generation for arbitrary regions with cracks. In: SIBGRAPI ’99: Proceedings of the XII-Brazilian Symposium on Computer Graphics and Image Processing, pp. 29–38, IEEE Computer Society
3. de Oliveira Miranda AC, Martha LF, Wawrzynek PA, Ingraffea AR (2009) Surface mesh regeneration considering curvatures. Eng Comput 25(2):207–219
4. Wu P, Houstis EN (1996) Parallel adaptive mesh generation and decomposition. Eng Comput 12(3–4):155–167
5. Lämmer L, Burghardt M (2000) Parallel generation of triangular and quadrilateral meshes. Adv Eng Softw 31(12):929–936
6. Larwood BG, Weatherill NP, Hassan O, Morgan K (2003) Domain decomposition approach for parallel unstructured mesh generation. Int J Numer Methods Eng 58:177–188
7. Ivanov EG, Andrä H, Kudryavtsev AN (2006) Domain decomposition approach for automatic parallel generation of tetrahedral grids. Comput Methods Appl Math 6(2):178–193
8. Głut B, Jurczyk T (2008) Domain decomposition techniques for parallel generation of tetrahedral meshes. In: Bubak M, van Albada D, Dongarra J, Sloot P (eds) Proceedings of the International Conference on Computational Science 2008. Lecture Notes in Computer Science, vol 5101, pp 641–650, Springer, Berlin, Heidelberg
9. Yılmaz Y, Özturan C, Tosun O, Özer AH, Soner S (2010) Parallel mesh generation, migration and partitioning for the Elmer application. Tech. Rep., PREMA-Partnership for Advanced Computing in Europe
10. Chen J, Zhao D, Huang Z, Zheng Y, Wang D (2012) Improvements in the reliability and element quality of parallel tetrahedral mesh generation. Int J Numer Methods Eng 92(8):671–693
11. Zagaris G, Pirzadeh SZ, Chrisochoides NP (2009) A framework for parallel unstructured grid generation for practical aerodynamic simulations. In: Proceedings of the 47th AIAA Aerospace Sciences Meeting, AIAA-American Institute of Aeronautics and Astronautics
12. Wu H, Guan X, Gong J (2011) ParaStream: a parallel streaming Delaunay triangulation algorithm for LiDAR points on multicore architectures. Comput Geosci 37(9):1355–1363
13. Topping BHV, Khan AI (1996) Subdomain generation for non-convex parallel finite element domains. Adv Eng Softw 25(2–3):253–266
14. Karypis G, Kumar V (1998) METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0. University of Minnesota, September
15. Karypis G, Schloegel K, Kumar V (2011) ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.2. University of Minnesota, April
16. Chrisochoides NP, Nave D (2000) Simultaneous mesh generation and partitioning for Delaunay meshes. Math Comput Simul 54(4–5):321–339
17. Linardakis L, Chrisochoides NP (2006) Delaunay decoupling method for parallel guaranteed quality planar mesh refinement. SIAM J Sci Comput 27(4):1394–1423
18. Rivara M-C, Calderon C, Fedorov A, Chrisochoides NP (2006) Parallel decoupled terminal-edge bisection method for 3D mesh generation. Eng Comput 22:111–119
19. Ito Y, Shih AM, Erukala AK, Soni BK, Chernikov AN, Chrisochoides NP, Nakahashi K (2007) Parallel unstructured mesh generation by an advancing front method. Math Comput Simul 75(5–6):200–209
20. Panitanarak T, Shontz SM (2011) MDEC: MeTiS-based domain decomposition for parallel 2D mesh generation. Proc Comput Sci 4(0):302–311. Proceedings of the International Conference on Computational Science 2011
21. Löhner R (2014) Recent advances in parallel advancing front grid generation. Arch Comput Methods Eng 21(2):127–140
22. Hodgson DC, Jimack PK (1996) Efficient parallel generation of partitioned, unstructured meshes. Adv Eng Softw 27(1–2):59–70
23. Khan AI, Topping BHV (1991) Parallel adaptive mesh generation. Comput Syst Eng 2(1):75–101
24. Wilson JK, Topping BHV (1998) Parallel adaptive tetrahedral mesh generation by the advancing front technique. Comput Struct 68(1–3):57–78
25. Topping BHV, Cheng B (1999) Parallel and distributed adaptive quadrilateral mesh generation. Comput Struct 73(1–5):519–536
26. Chrisochoides NP (2005) A survey of parallel mesh generation methods. Tech. Rep. SC-2005-09, Brown University
27. Okusanya T, Peraire J (1996) Parallel unstructured mesh generation. In: Proceedings of the 5th International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, pp. 719–729, Mississippi State University
28. Chernikov AN, Chrisochoides NP (2006) Parallel guaranteed quality Delaunay uniform mesh refinement. SIAM J Sci Comput 28(5):1907–1926
29. Lo SH (2012) Parallel Delaunay triangulation—Application to two dimensions. Finite Elem Anal Des 62:37–48
30. Lo SH (2012) Parallel Delaunay triangulation in three dimensions. Comput Methods Appl Mech Eng 237240:88–106
31. Chernikov AN, Chrisochoides NP (2005) Parallel 2D graded guaranteed quality Delaunay mesh refinement. In: Proceedings of the 14thInternational Meshing Roundtable, (San Diego, United States), Sandia National Laboratory
32. De Cougny HL, Shephard MS (1999) Parallel volume meshing using face removals and hierarchical repartitioning. Comput Methods Appl Mech Eng 174(3–4):275–298
33. Löhner R (2001) A parallel advancing front grid generation scheme. Int J Numer Methods Eng 51(6):663–678
34. Freitas MO, Wawrzynek PA, Cavalcante-Neto JB, Vidal CA, Martha LF, Ingraffea AR (2013) A distributed-memory parallel technique for two-dimensional mesh generation for arbitrary domains. Adv Eng Softw 59:38–52
35. Batista VHF, Millman DL, Pion S, Singler J (2010) Parallel geometric algorithms for multi-core computers. Comput Geom Theory Appl 43(8):663–677
36. Kohout J, Kolingerová I, Žára J (2005) Parallel Delaunay triangulation in E^2 and E^3 for computers with shared memory. Parallel Comput 31(5):491–522
37. Angel E (2008) Interactive computer graphics–a top-down approach using OpenGL, 5th edn. Addison Wesley, Boston