Parallel computing for 2D / 3D meshing manipulation [1]

Paul LAFOIX-TRANCHANT, Antoine OLEKSIAK
Supervisors: Fabrice Jaillet, Florence Zara
POM 2018-2019

Abstract: The purpose of this project is to parallelize an existing library in order to perform 2D/3D meshing in real-time for a Virtual Reality environment.
Keywords: mesh, 2D/3D, real-time, parallel algorithm, Intel TBB, OpenMP**.**

1. Introduction

a. Context

The ECOS project "Simulators for the Learning of Gestures of Childbirth" aims at the development of a Virtual Reality environment allowing the realization of a simulator for learning the medical gesture of childbirth. This simulator consists of a numerical simulation coupled to a physical device. One of the major issues of the digital part is the complexity of the interactions between the objects. These may be internal, such as the contact between the uterus and the fetus during a contraction, or external as the pressure of the forceps on the fetal head.

In this project, one of the research tracks concerns the use of so-called mixed meshes, combining different types of elements (triangles, quadrangles in 2D or tetrahedrons, hexahedrons, prisms and pyramids in 3D). This geometric model, coupled with an adapted simulation technique, should make it possible to manage complex scenes compatible in real time with such a virtual reality simulator.

In order to do that, Mesher_Roi, a Quality Mixed-Element Mesh Generation and Refinement program, has been developed by Claudio Lobos (Departamento de Informática, UTFSM, Chile) and Fabrice Jaillet (IUT Lyon 1, LIRIS). This mesh generator will allow you to create a mixed-element mesh starting from a boundary 2D mesh (Polyline) composed of edges or 3D mesh from a 3D shape composed of triangles.

b. Objective

The existing library is currently focused on 2D in order to perform a feasibility study and a parallelization solution, in view of 3D, certainly more complex but very similar, and which corresponds to the intended application. The 2D mixed-element mesh generator is not fast enough for the future 3D real time constraint or large mesh, thus the mesh generation requires optimizations.

The objective of our POM project is to speed up the meshing and remeshing process using parallelism on the CPU. For this purpose, we have to choose a complete solution to parallelize the existing algorithms: the parallel strategy and the programing language.

In the following, we will present a general analysis of the existing library then focus on a more precise part that we will try to parallelize. We will explain how we've proceed and our results.

2. Analysis of Mesher_Roi 2D library

a. Purpose

The aim of the Mesher_Roi 2D library is to create a surface with quality elements by taking a polyline as input, with aim to make simulations (with external tool) of the deformation of this surface.

[1] English version of the report upon request of the supervisors, to ease transmission with the Chilean partner

b. How it works

An input surface mesh must be unfolded (no self intersection) and the normal of the edges must be pointing outside (Fig. 1). Features correspond to sharp angles between 175° and 185°.

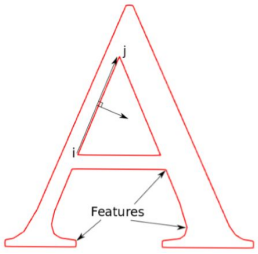

Figure 1: An example of complex polyline with holes and sharp features. This corresponds to the input to be meshed.

This mesh generator is based on the Quad-tree technique, which consists in recursively split a **Quadrant** in a finite number of equivalent childs (see Algorithm 1). For instance, if the **Quadrant** is a square, to refine it one level means that it will be replaced by four new **Quadrants**. All of them will be sons of the replaced **Quadrant** in the Quad-tree structure. In our case, **Quadrants** will be continuously split until a maximum provided level is reached.

Algorithm 1: Generation process

```
Result: A mixed-elements mesh
/* Generate quadtree:                                    */
repeat
    foreach Quadrant do
        Subdivide Quadrant;
        foreach new Sub-Quadrant do
            if Intersects Input or Is Inside then
                Insert Sub-Quadrant
            end
        end
    end
until desired Refinement Level;
Create balanced quadtree;
Apply Transition Patterns;
```

Two options are available:

- Refine all quadrants to a desired level **N** (they will be refined at least once).
- Refine surface quadrants to a desired level **N**, the splitting operation will be performed over each quadrants that intersects a section of the input.

Theses two options can be combined and in both cases quadrants lying completely outside the input will be removed. Fig2. shows the effect of the refinement applied on all **Quadrants**.

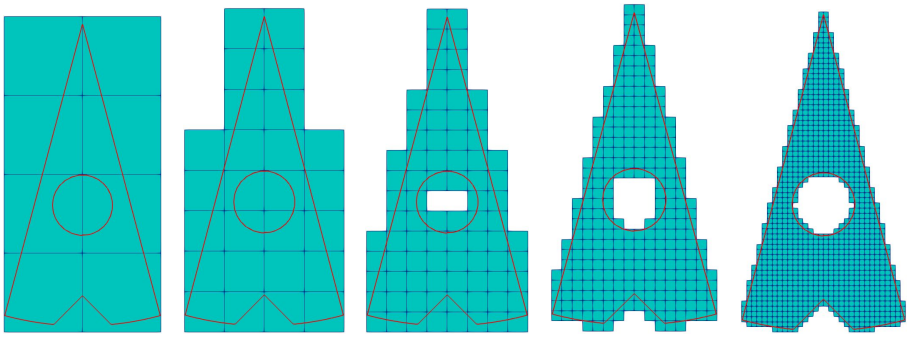

Figure 2: Effect of the refinement level applied on the complete structure (refine all N=1..5).

We can note that large meshes can be produced with a small value for the desired level. For instance, if the starting point is only one cube and we set **N**=10, the code may produce up to 1,048,576 elements. Of course, all these switches might be combined (as in Fig. 3) to produce flexible tools and comply with the user requirements.

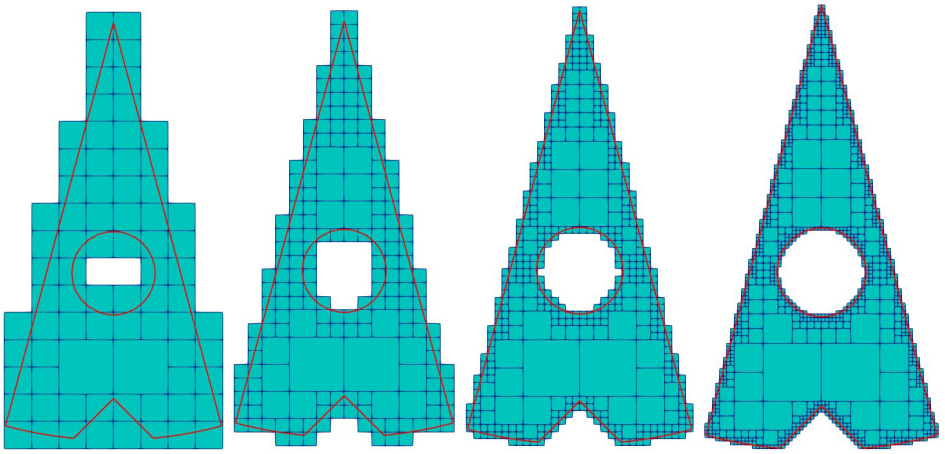

Figure 3: Effect of combined refinement levels, with level 2 applied on the complete structure and incremental refinement only applied on surface quadrants (N=3..6).

A Quad mesh is said balanced when there is at most a difference of 1 level of subdivision between any two adjacent elements. In practice, this is determined by checking whether a **Quadrant** contains at least one **QuadEdge** that is subdivided twice. Thus, for each **Quadrant**, every **QuadEdge** is checked. If one of them is subdivided, then the 2 sub-edges must be checked (Fig. 4).

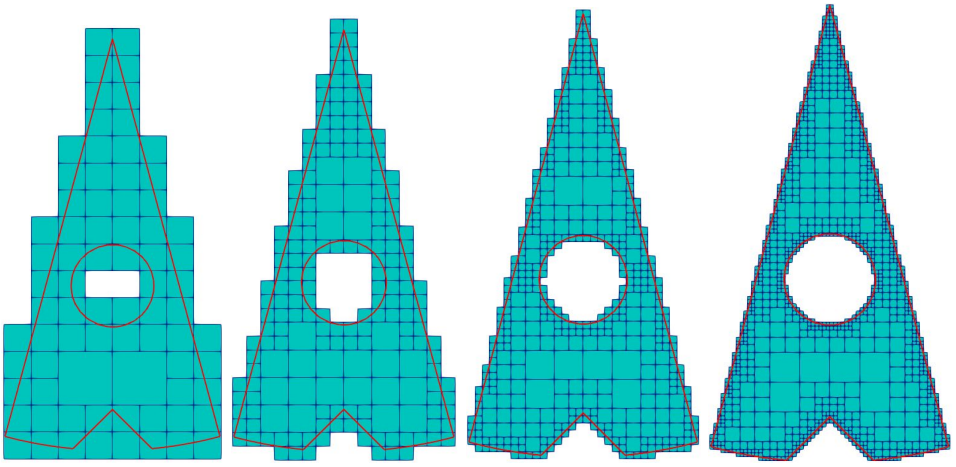

Figure 4: After the balancing step of previously refined mesh (Fig. 3). Note that in the first case, nothing is to be done.

Now the mesh is balanced, we need to handle transitions between **Quadrants**, to produce a conformal mesh, when the elements exactly share nodes and edges (Fig. 6). This technique is known to save time during the computation, and also to avoid node interpolation errors. If 2 adjacent **Quadrants** have the same refinement level, then there is nothing to do. Otherwise, the less refined **Quadrant** must be treated. Again, in practice, this is determined by checking whether a **Quadrant** contains at least one **QuadEdge** that is subdivided. For the Transition, the chosen technique was to apply a Pattern corresponding to the number and relative position of the refined **QuadEdges**. In 2D, there are only 6 cases to consider (Fig. 5).

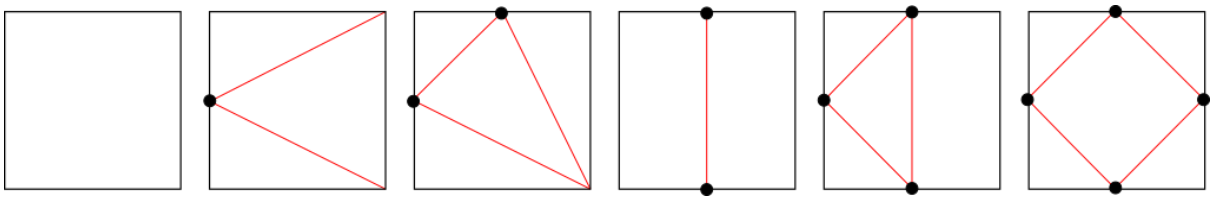

Figure 5: All possible configurations for transition patterns in 2D. Black dots represent an edge subdivision, while red lines draw the internal edges of the Quadrant sub-elements.

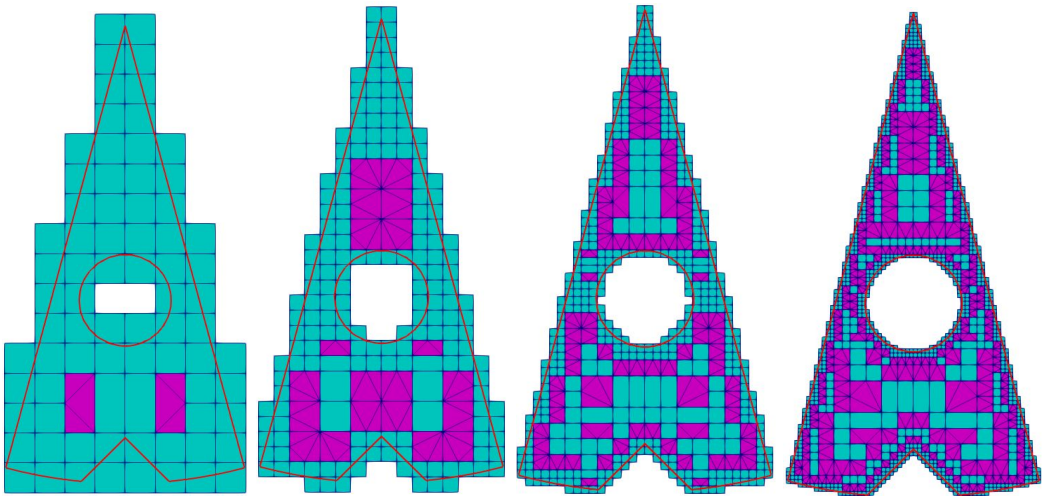

Figure 6: After the transition step. Note that the more difference between levels, the more transition patterns are used.

In the Mesher_Roi 2D library, there are some important structures to consider:

Polyline:

- corresponds to the input surface;
- is composed of a vector of 3D Points, and their associated angle.

Quadrant:

- is composed of a *pointindex*, vector of 4 corner indices in a vector of **MeshPoint** (**MeshPoint** is embedding **3DPoint**, as well as some information describing its state).
- When the **Quadrant** is subdivided into non quadrant elements (in Transitions for example), it contains a vector of *sub_elements*, each one described by a vector of indexes as presented right above.
- It also contains a list of intersecting **PolyEdges**, and intersecting **Features**. This information is useful to speed up the inside/outside discrimination for the **Quadrant** or to better handle the boundary and surface elements, for example.

QuadEdge:

- corresponds to the edges of the quadrants;
- is defined by 3 indices, the 2 extremities and potential *midpoint* when this edge is split.

MeshPoint:

- that is embedding a **3DPoint**, as coordinates of the **Quadrants** corners, referenced as indexes.
- **MeshPoint** are conversely connected to **Quadrant** by an index map of *elements*, referencing all the **Quadrants** having this point as corner.

- it is also the support for some crucial processing information on its *state*, merged in a single byte: the point is inside, has been projected, is representing a feature, has been previously checked, etc.

These three structures will be used together along with the **Polyline** by the **Mesher**, that will construct them gradually:

- a vector of **MeshPoint**
- a vector of **Quadrant**
- a set of **QuadEdge**, where QuadEdge are sorted by the two indices of the extremities

Now that all the important information about the library has been introduced, it is time to see how we can parallelize it.

3. Parallel version with mutex on critical sections

In this section, we will focus on the quadrant splitting step in order to analyze the concurrency problem raised if we decide to simply launch multiple thread on the sequential algorithm. We will see that there are multiple critical sections in the main loop and in the visitors used in this loop.

a. Existing sequential algorithm of Mesher_Roi 2D

i. Structures

The sequential implementation of Mesher_Roi 2D uses the following data structures:

- The vector of **Quadrant,** called *tmp_quadrants*. Note that the loop will iterate over and remove them once processed.
- A list of **Quadrants,** called *new_quadrants,* where all the new quadrants will be stored.
- The vector of **MeshPoints** is called *points*.
- A list of **Point3D** called *new_pts* where all new points will be stored.
- The set of **QuadEdges** is called *QuadEdges*.

Now, let see how the Algorithm1 is implemented in sequential.

ii. Main loop

At the beginning of each level, *new_quadrants* and *new_pts* are cleared, and at the end, content of *new_quadrants* and *tmp_quadrants* are swapped and the content of *new_pts* is inserted in the vector of **MeshPoint**.

Each **Quadrant** that needs to be subdivided use the **SplitVisitor** and each new **Quadrant** is inserted in *new_quadrants* (and also those which have reached the desired level of refinement).

The **SplitVisitor** has multiple critical sections so we need to deepen the subject.

iii. SplitVisitor

It reads *points* structure, inserts in *new_pts* and reads / inserts / removes in *QuadEges* structure.

The point of this visitor, as its name suggests, is to split a given **Quadrant**. This operation creates up to 14 **QuadEdge** and 5 **Point3D** (Fig. 7). Here is its content:

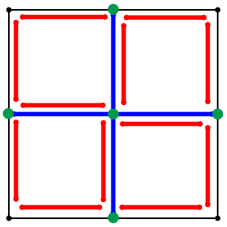

*Figure 7: Original **Quadrant** in black, new edges created in blue and red, new points created in green.*

First of all, for each edge of the **Quadrant**, it checks if the edge had already been split by searching this **QuadEdge** in the global structure *QuadEdges*. If a index of *midpoint* for this **QuadEdge** is already defined (i.e. different from 0), this index is stored in order to be used later, if not this edge has to be split. This split operation is done in this order for each of the edges:

1. Get the future *midpoint* index (it will be added in *new_pts* and all content of *new_pts* will be added in *points* so its future index is *points* size plus *new_pts* size).
2. Update the **QuadEdge** *midpoint* with the computed future index.
3. Erase the corresponding **QuadEdge** in *QuadEdges* as we cannot modify an element of a set.
4. Insert the **QuadEdge** with the correct *midpoint* in *QuadEdges*.
5. Create and insert the two new split **QuadEdge** with the new *midpoint* index.
6. Insert the new **Point3D** (corresponding to the *midpoint*) in *new_pts*.

Then, 6 edges (which represent the inside edges when you split a **Quadrant** in Fig. 8, the 2 blue ones and the 4 red ones which have the green center point in commun) are created and inserted in the set and 1 **Point3D**. Plus a vector containing the **Point3D** indices of the 4 new quadrants is returned in order to build the **Quadrants** outside the visitor.

b. Comparison between Intel TBB and OpenMP

In order to parallelize the library, we gathered information about multi processing APIs. Among all these available APIs, Intel TBB and OpenMP stand out: Intel TBB provides concurrent structure and seems to be more useful for nested loop; OpenMP seems easier to use and has a lot of documentation compared to Intel TBB.

Consequently, to choose the best method according to the datasets that are used in the mesh generator, we first developed a program that uses the visitor pattern (as in Mesher_Roi 2D) to do simple computations on different data structure, such as *vector* (Fig. 8), and *list* (Fig. 9).

After comparison, it shows up that openMP perform better performances than Intel TBB on every data structures. However, the advantage of Intel TBB is the concurrent data structure it provides. For example, a performant *concurrent_unordered_set* is provided. It can be useful to replace the set of edges in Mesher_Roi 2D which is the biggest global structure, which need to be read and modified frequently.

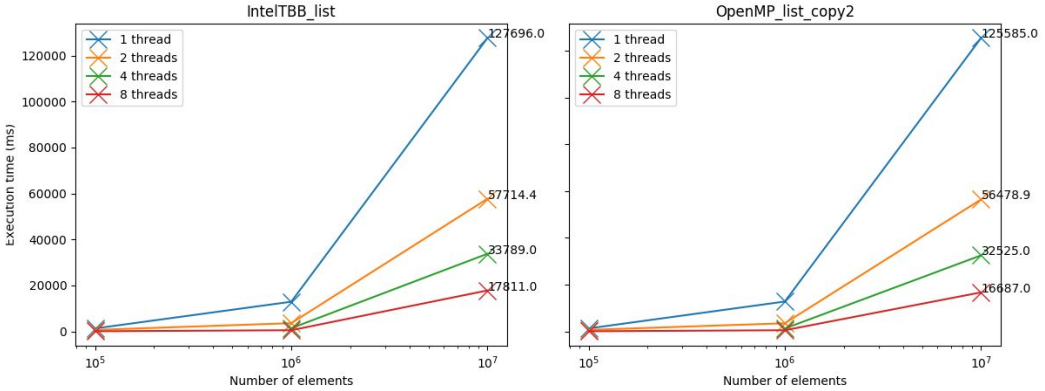

Figure 8: Comparison of computation on list with Intel TBB and openMP.
Notice that openMP converts list into vector.

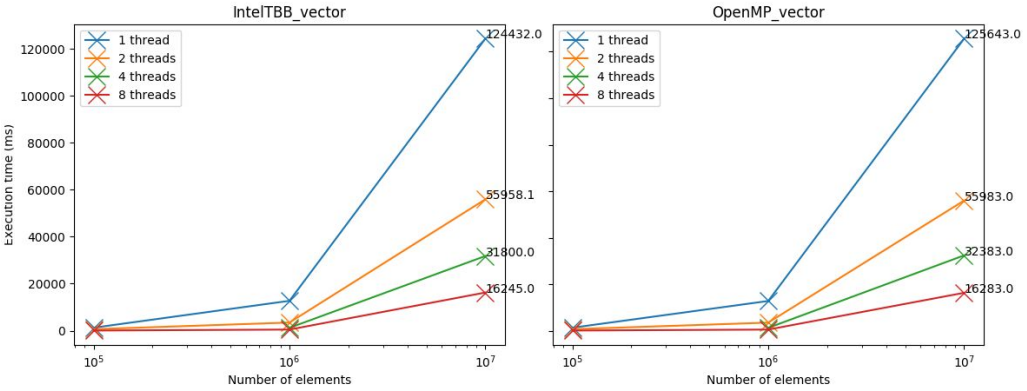

Figure 9: Comparison of computation on vector with Intel TBB and OpenMP.

After that, as we cannot found a good reason to use an API rather than another, we decided to split the work in two: one was in charge of the implementation in Intel TBB and the other in OpenMP.

c. A parallel solution using multi-threading

Firstly, we have simply protected all the critical sections with mutex, that is around the uses of structure *new_pts*, *QuadEdges*, *tmp_quadrants* and visitor as described above. As expected, computational time was very bad.

Secondly, we tried to remove mutex by replacing each structure by concurrent structure provided by Intel TBB in order to perform locks in portion of structure instead of all. However, **SplitVisitor** portion of code, described above, still needed mutex protection. This method improves computational time but it was still bad comparing to sequential.

So, we decided to search a way to remove the remaining mutex. We used a concurrent unordered set which allows us to insert an edge in concurrency. We've searched a way to update the *midpoint* without having to erase and insert again the edge, as the structure does not provide a thread safe erase method and the *midpoint* is not used by the set sorting. We found the attribute *mutable* which allows us to modify the *midpoint* of a **QuadEdge** in a set. There is one problem left, two (or more) threads can check for the same edge at the same time, thus the edge can be splitted multiple times. As a result, multiple **Point3D** corresponding to the same *midpoint* will be created, the **QuadEdge** *midpoint* will be overridden and the two new **QuadEdge** won't have the same used index as their parent **QuadEdge**. To avoid that, we decided to use a mutex in the method that updates the *midpoint* in **QuadEdge** and this method allows a modification of the *midpoint* only if it has not been defined. We also needed to reorder the operation done in sequential algorithm.

The split edge operation now looks like this:

1. Insert the new **Point3D** (corresponding to the *midpoint*) in *new_pts* (originally 1, 6).
2. Update **QuadEdge** *midpoint* in *QuadEdges* with the index resulting from the insert (originally 2, 3, 4).
3. Get the *midpoint* of the corresponding edge in *QuadEdges (in case another thread modified it before this one)*.
4. Create and insert the two new split **QuadEdge** with the *midpoint* index (originally 5).

Useless points (unused in the final solution) will be inserted but this will speed up the algorithm and this method finally improves computational time compared to sequential (see results below).

d. Results

Here, we present the time results of 12 levels of refinement applied on all **Quadrants** and on surface **Quadrants** (Fig. 10). At the end of refinement apply on all there are 4,095,572 **Points**,

13,634,086 **QuadEdges**, 4,057,578 **Quadrants**. And at the end of refinement apply on surface there are 106,819 **Points**, 320,520 **QuadEdges** and 55,533 **Quadrants**.

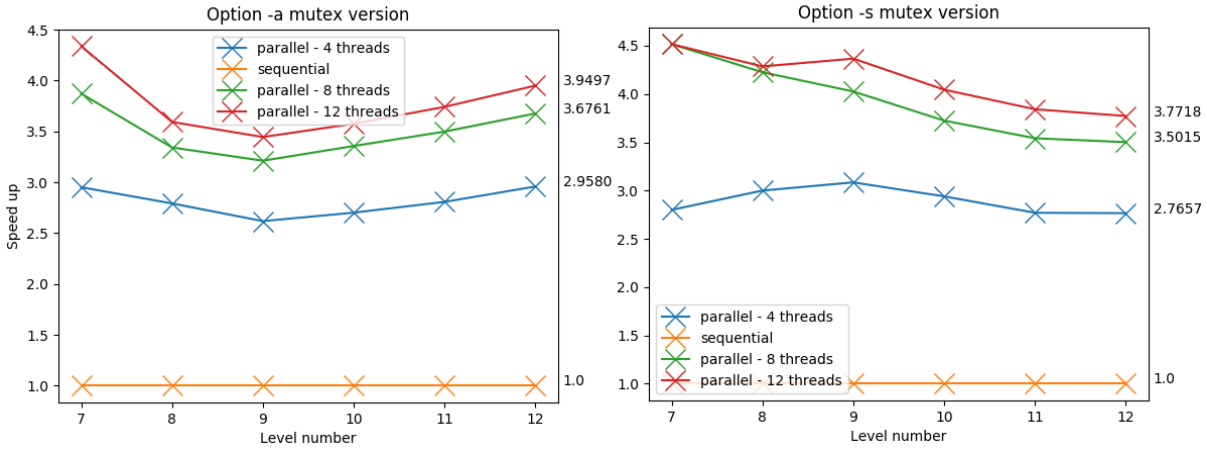

Figure 10: Speed up results of the mutex version of refinement in [7, 12] to all (left) and only for surface (right).

With refinement on all at level 12, there are only 7 useless **Points** created that are useless (that will anyway be removed in the final solution). Only 8 with refinement on surface at level 12.

This algorithm speeds up time at any levels. But, in order to avoid critical sections that slow the entire process, and to avoid the useless creation of points, a version with a parallel reduction has been developed.

4. Parallel version with a reduction

a. Method

It consists of letting the threads work in only local variables (accumulation process), and then let threads do the reduction (i.e. store in the global structures **QuadEdges**, **Quadrants** and **Points**) all the new points, new edges and new quadrants created by the threads.

The difficulty here was to detect if a point created by a thread is the same as another point created by another thread. Indeed, during the accumulation process, all the workers threads start creating points from the number of points before the accumulation part. Thus, **Point3D** which have different location (*resp*. the same) may have the same (*resp*. another) local index causing the locals **QuadEdge** and **Quadrant** to have indices that will need to be changed during the reduction step.

An example is presented in Fig. 11. The green thread start splitting the quadrant at the left side, and the blue at the right side. The reduction process needs to identify that point number 7 and 9 are the same, as well as edges 1-2-7 and 1-2-9, 1-7-0 and 1-9-0, 2-7-0 and 2-9-0. It also needs to update the good global index.

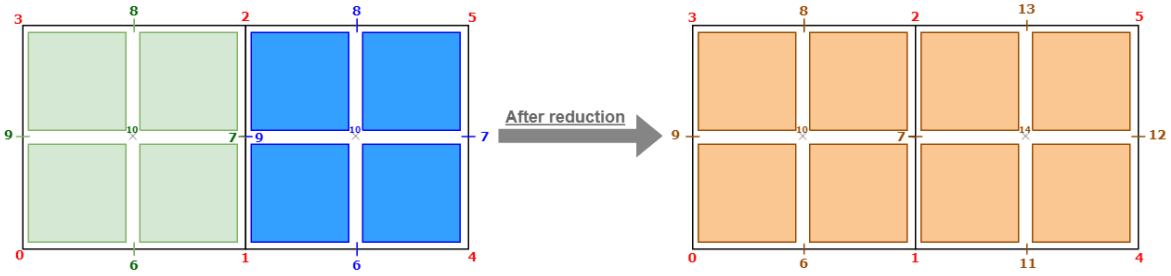

Figure 11: Scheme that shows the threads numbering of new points. In red, index of points that were here before the splitting process. In green the work done by the first thread, in blue by the second one. In brown the result of the reduction.

In order to do that, a first reduction algorithm has been written, presented in Algorithm 2.

Algorithm 2 Reduction of parallelized accumulation

```
First fill the map that link local index to global
nbPointsBeforeJoin ← points.size()(constant)
for each thread T do
  taskToGlobal ← map < int, int >
  for each point in T.accumulatePoints do
    if point not in new_pts then
      new_pts.add(point)
      map_new_pts[point] ← totalNbPoints
      totalNbPoints ← totalNbPoints + 1
    end if
    // At this time, map_new_pts contains real index of the global index of
    the local point
    index ← point.localIndex + nbPointsBeforeJoin
    taskToGlobal[index] ← map_new_pts[point]
  end for

  for each edge in T.accumulateEdges do
    for each index in edge do
      if index ≥ nbPointsBeforeReduce then
        // we need to update to global index
        index ← taskToGlobal[index]
    end if
  end for
  new_edges.add(edge)

  for each quad in T.accumulateQuad do
    for each index in quad.pointIndex do
      if index ≥ nbPointsBeforeReduce then
        // we need to update to global index
        index ← taskToGlobal[index]
    end if
  end for
  new_quad.add(quad)
end for

return new_pts, new_edges, new_quad
```

We have tested multiple implementations before successfully reduce the computational time comparing to sequential.

Our first tries were with the parallel reduce function implemented in Intel TBB which, given a class (called a *task* with a split, body and a join) and a range on a structure, perform the reduction. It recursively splits the range into subranges, uses the available worker thread to perform the body method and for each such split of the body, it invokes method *join* in order to merge the results from the bodies. This method was longer than the sequential as the *join* method was very time consuming.

After this failure, we have decided to make our custom reduction in order to avoid performing intermediate join. As such, we decided to elect a master task, equally distribute to all tasks the quadrants to be processed. And once all the tasks are completed, the master task performs the join for each task. This method was still longer than the sequential.

However, as all tasks are done, we have tried to parallelize the join method. The most critical part is the one that deals with the points. So we have decided to run all the points parts with the master task and then perform the edges and quadrants parts in parallel using thread-safe structure provided by Intel TBB. Finally, this method improves computational time (see results below) but only after a certain number of **Quadrant** to process (around 50,000).

b. Results

Here is the time results of 10 levels of refinement applied on all **Quadrants** and on surface **Quadrants** (Fig. 12). At the end of refinement apply on all there are 266,291 **Points**, 883,200

QuadEdges, 255,663 **Quadrants**. And at the end of refinement apply on surface there are 26,493 **Points**, 79,540 **QuadEdges** and 12,615 **Quadrants**.

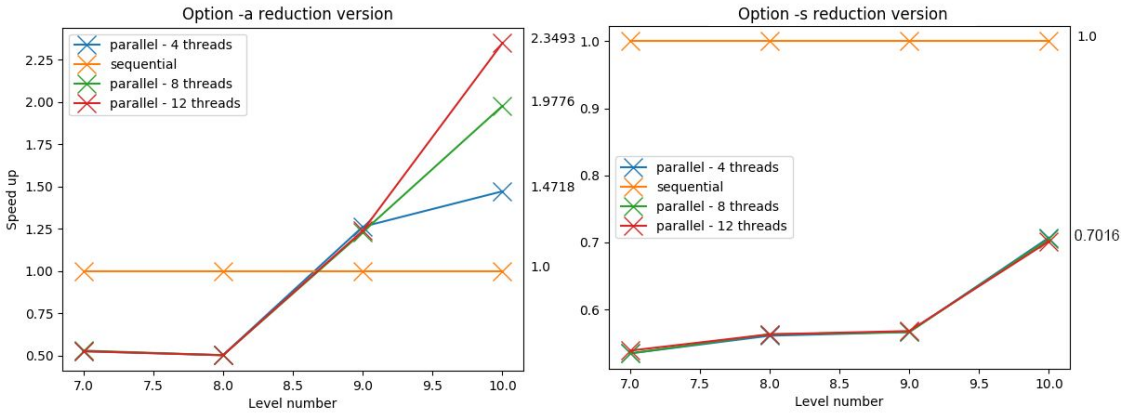

Figure 12: Speed up results of the reduction version of refinement in [7, 10] to all (left) and only for surface (right).

As you can see, the reduction version is not efficient when they are to few **Quadrants** to process (before level 8 in -a). The join part is the slowest part of the reduction algorithm as only one thread is doing the points join, and that is why this method is faster than the sequential version only when there are enough quadrants (sequential time is much higher than just the join part). Thus, increase the number of threads will speedup the algorithm until the benefit is lower than the time to do a join on the new threads.

5. Conclusion

The "mutex" version is actually more performant than the reduction one especially when there are less than 50,000 **Quadrants**. Besides, there is only small changes between the sequential and parallel version whereas in the reduction, there are also small changes, but there are a lot more code added for the joint part which make the code less maintainable.

However, if the sequential version is needed to be executed, even if there is only one thread that executes the whole code, it will be slowed by the locks that need to be acquired by concurrent structures and *midpoint*.

This project taught us the difficulty of parallelisation and comprehension of complex library.

6. Perspectives

A promising first study which needs to be improved in view of a 3D implementation for gestures of childbirth. Here are some trails to explore in order to speedup the algorithm.

- Build a quadtree structure to represent all the **Quadrants** which will be useful in the reduction in order to replace / erase / insert a whole subtree instead of running through all the **QuadEdges** and **Quadrants**.
- Make the reduction only when all **Quadrants** are reached their desired levels. The reduction is the most costly part but this needs to be done wisely. Indeed, if only the **Quadrants** that intersect the input must be refined, some threads may have much less **Quadrant** to split which would be a lost of time. Furthermore, the actual reduction algorithm is based on merge two results with only one depth level difference.
- Use unique ID for points which would produce the same ID if multiple thread create the same point. This will speed up time for the reduction as we will no longer need to map points. This will also avoid the needing to protect the *midpoint* in the mutex algorithm (which will really speed up the algorithm). We think this is the best trail to explore to improve the computational time.