Parallel computing for 2D/3D meshing manipulation

Paul LAFOIX-TRANCHANT, Antoine OLEKSIAK

Lyon university

Supervised by Fabrice JAILLET and Florence ZARA

Context

- 2D / 3D Mesh refinement (Region Of Interest)
- Real time simulation

Objective

- Analyze the existing library
- Find data structure to optimize parallelization
- Speed up the meshing process

Mesh construction

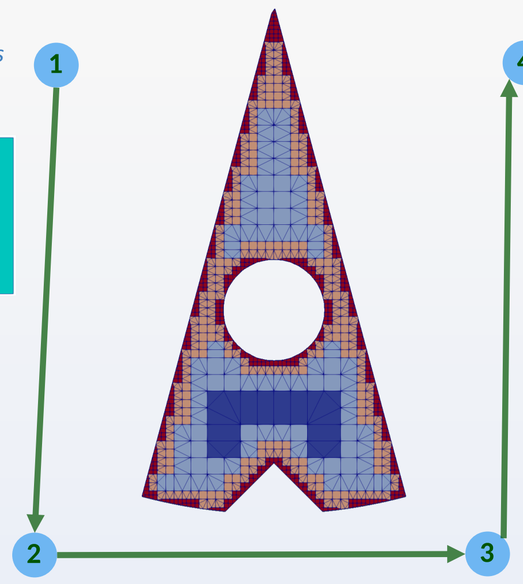
1. Compute the Bounding Box

Create the minimal number of initial quadrants that contains input surface

2. Generate Quadrants

Process :

Repeat until desired Refinement Level
  For each Quadrant :
    If the quadrant intersects the chosen region
      Split the quadrant into 4 new identical quadrants.
    Remove those which are outside.

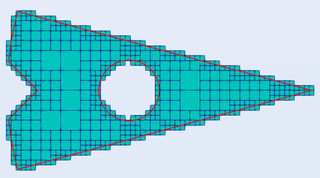


3. Create a balanced structure

(i.e. at most 1 subdivision level between two adjacent quadrants)

For each Quadrant :
  if the quadrant has a subdivided edge (midpoint != 0)
  check if the subdivided edge are subdivided

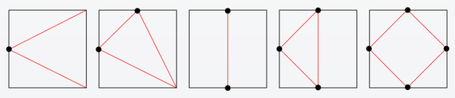
Balanced with surface region



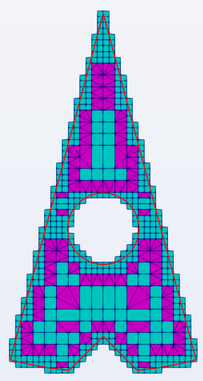
4. Apply the transition patterns

(i.e. to produce a conformal mesh)

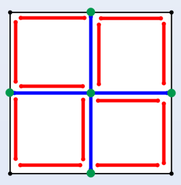
The transitions patterns



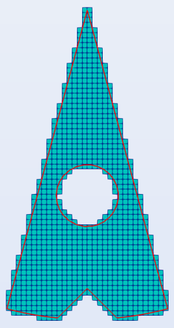
Transitions patterns applied on output of meshing on surface region



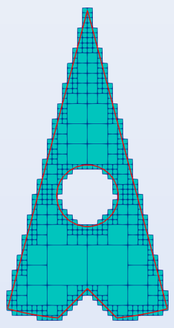
Splitting process :
in red and blue new edges
in green new points



Split all region



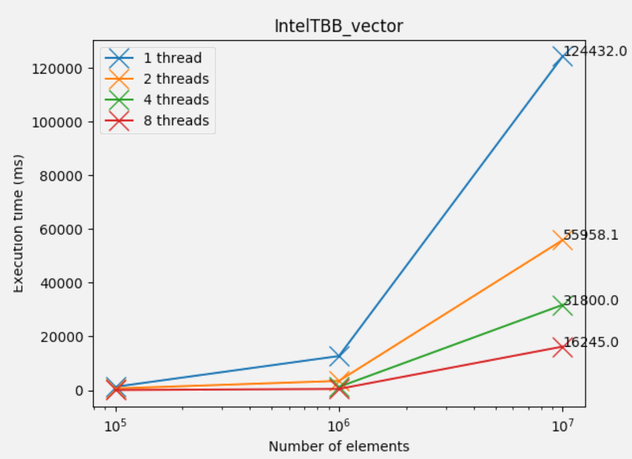
Split surface region



Libraries for C++ parallelism

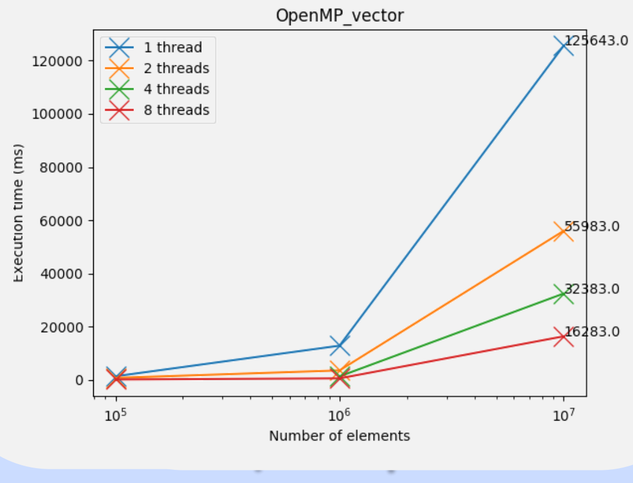
- IntelTBB

\+ Thread-safe containers
\- Only few examples



- OpenMP

\+ Pre-processing directives



Parallelize the mesh construction

Mutex version

- Protect the critical regions with different mutex.
- Use of concurrent data structures, such as tbb::concurrent_unordered_set to replace the set of QuadEdges.

Critical regions

- read/write in concurrency in the set of quadedge = one edge could be split twice
- Write new points and new quadrants

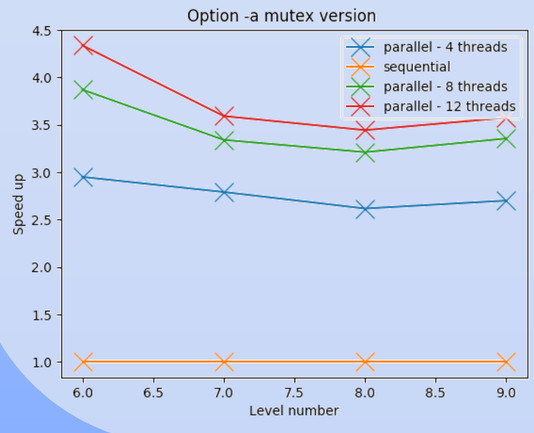
Problems

- It is not possible to modify an element in a set (eg. In the set of QuadEdges) because of the hashtable used to sort elements. However, it is possible to modify a mutable attribute of an element if it's not used for sorting.

Results

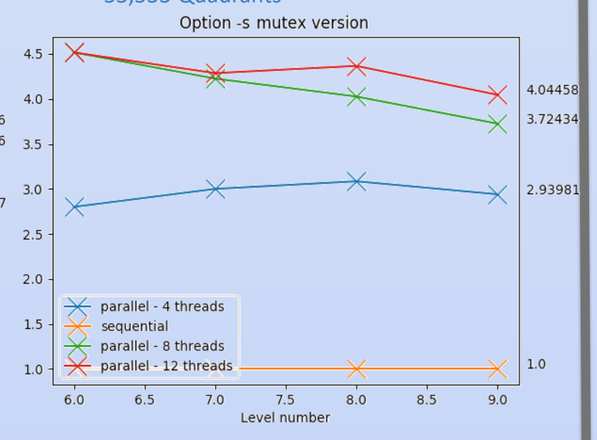
Refinement applied on ALL Quadrants

At the end :
  4,095,572 points, 13,653,086 QuadEdges and 4,057,578 Quadrants



Refinement applied on SURFACE Quadrants

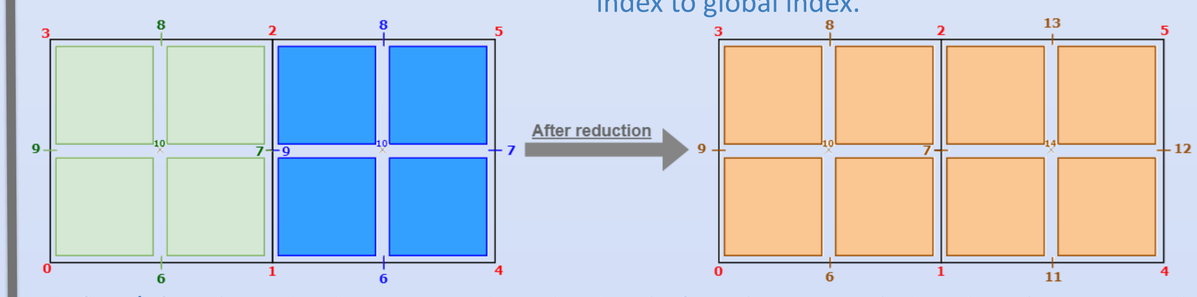
At the end :
  106,819 points, 320,520 QuadEdges and 55,533 Quadrants



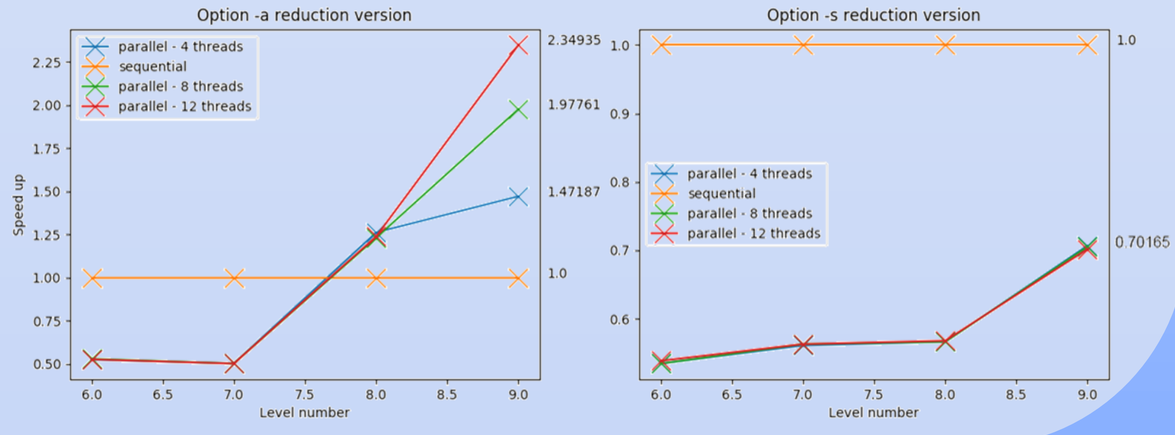
Reduction version

- No more critical regions.
- All threads have their own copy of Edges, Points and Quadrants filled in the accumulation part.
- The reduction part is done by a single thread and creates the final Edges, Points and Quadrants.
- Detection of identical points and update the local index to global index.

```
for (int rl=0; rl<desiredLevel; ++rl) {
    #pragma omp parallel
    {
        int tn = omp_get_thread_num();
        accumulation(newPts[tn], newEdges[tn], newQuad[tn]);
    }
    //Single thread
    reduction(newPts, newEdges, newQuad);
}
```



Before/after the reduction process. In red points before the accumulation done by green and blue threads. Notice that they start creating points at index 6 (the number of points before the accumulation)



Conclusion

- The mutex version is more performant than the reduction, even for high number of quadrants (> 10^9). The reduction part is done sequentially and should need optimizations
- Implementations avoiding critical sections produce a lot of new code, and it makes the whole project less maintainable
- Parallelism is not easy !

Perspectives

- Build a quadtree structure to represent all the Quadrants
- Speed up the reduction part of the version with reduction with a better algorithm and/or the use of parallelism
- Make the reduction process only when the desired refinement level is reached