Parallel computing for 2D/3D meshing manipulation

Paul LAFOIX-TRANCHANT, Antoine OLEKSIAK

Lyon university

Supervised by Fabrice JAILLET and Florence ZARA

Context

- Mesh generation and adaptation with quality mixed elements
- Real time simulation and deformation

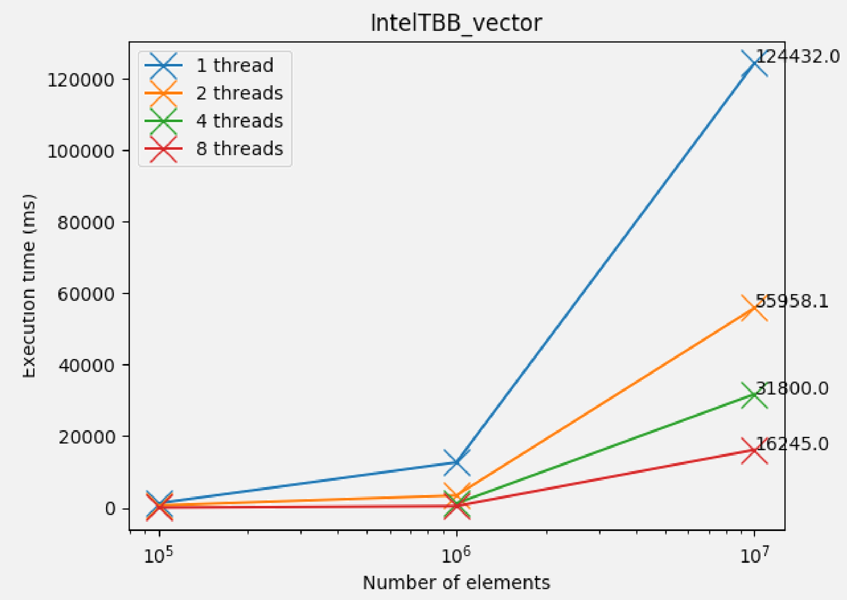
Objective

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

Tools : Libraries for C++ parallelism

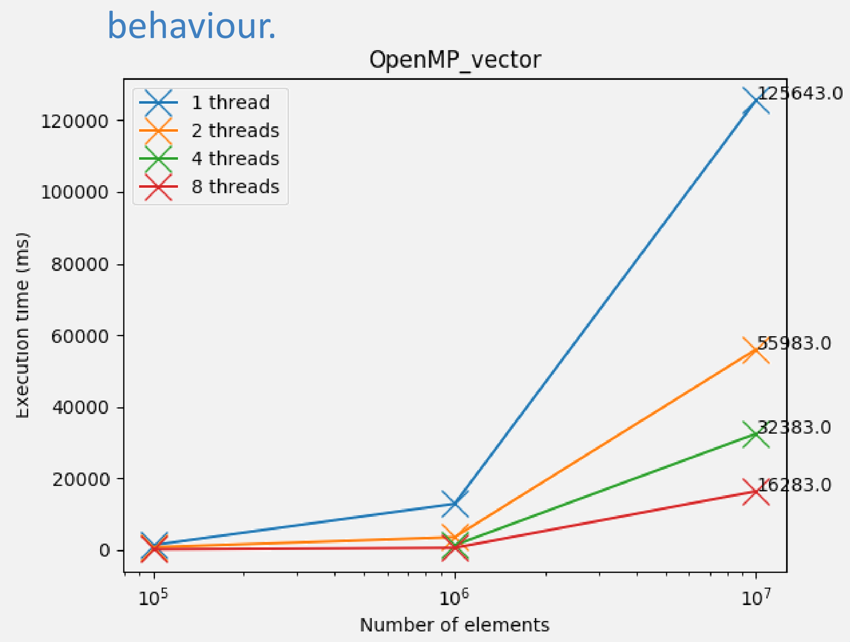
1) IntelTBB

A C++ template library developed by Intel that provides tasks, low-level synchronization primitives, concurrent containers, and is compatible with other threading packages.



2) OpenMP

An API that runs on most platform and provides a set of compiler directives, library routines and environment variables that influence run-time behaviour.



Mesh generation

1. Compute the Bounding Box

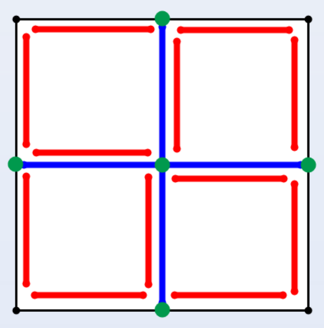
(i.e. create the minimal number of initial quadrants that contains the input surface)

2. Generate Quadrants

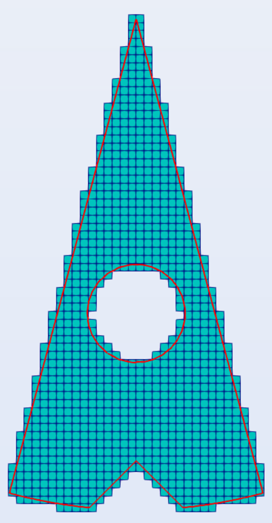
Process :

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.

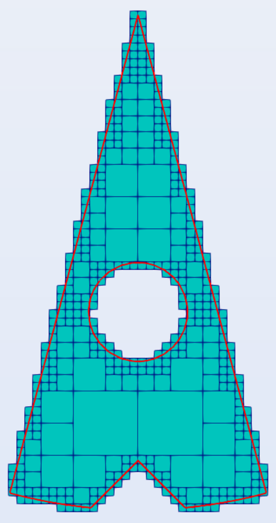
Splitting process :
in red and blue new edges, in green new points

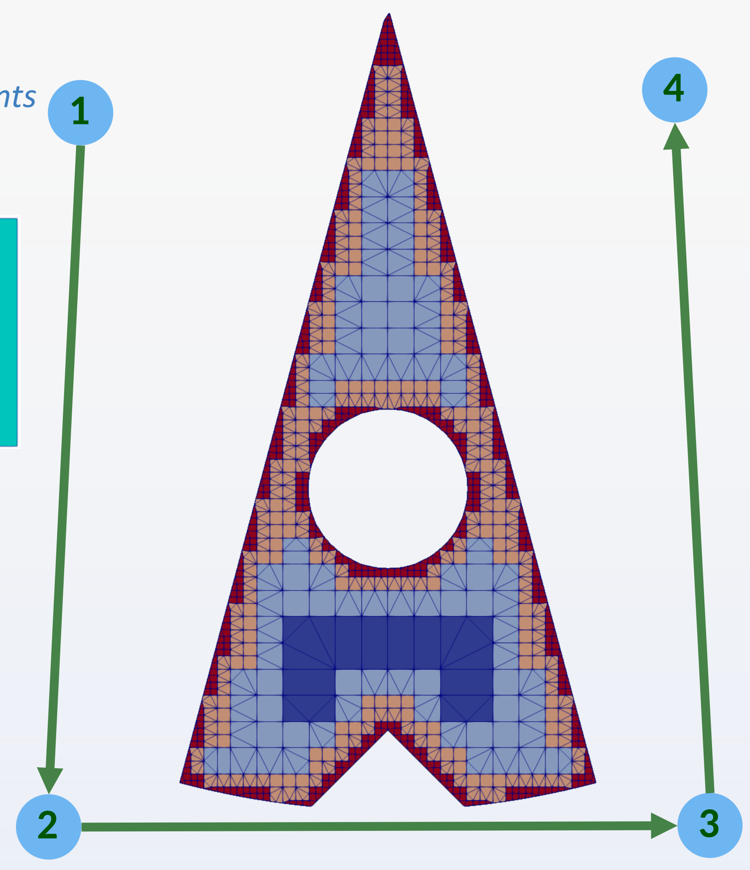


Split all region (-a)



Split surface region (-s)



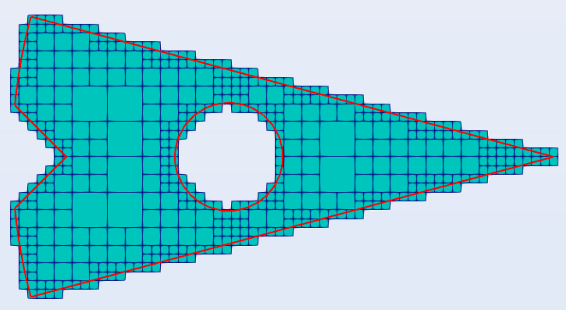


3. Create a balanced structure

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

Process :

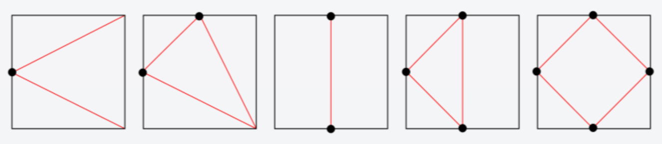
For each Quadrant q :
  if one neighboring quadrant has a subdivision level >= q.level + 2 :
    Split q into 4 new quadrants



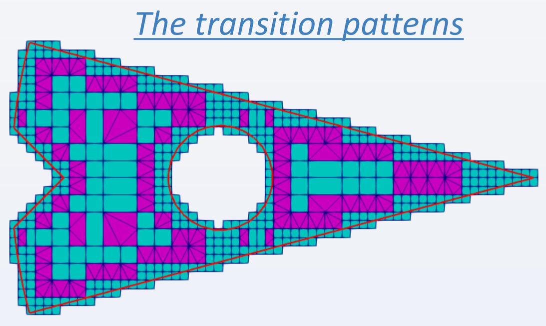
Balanced with surface region

4. Apply the transition patterns

(i.e. to produce a conformal mesh, create sub-elements following a pattern in a Quadrant if he has a refinement level < adjacent Quadrant)



The transition patterns



Transition patterns applied on output of the meshing on surface region

Parallelize the mesh generation

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 by 2 neighboring quadrants
- Write new points and new quadrants

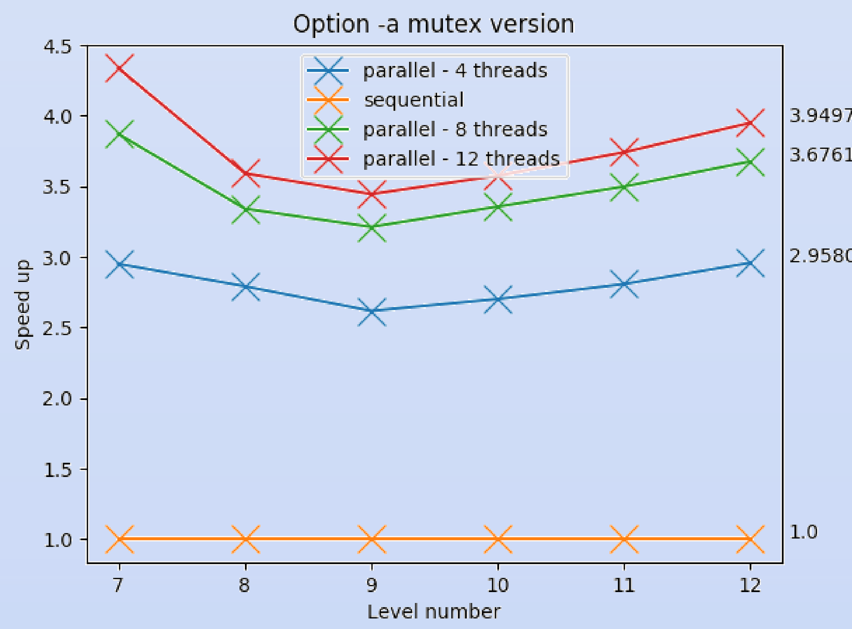
Problems

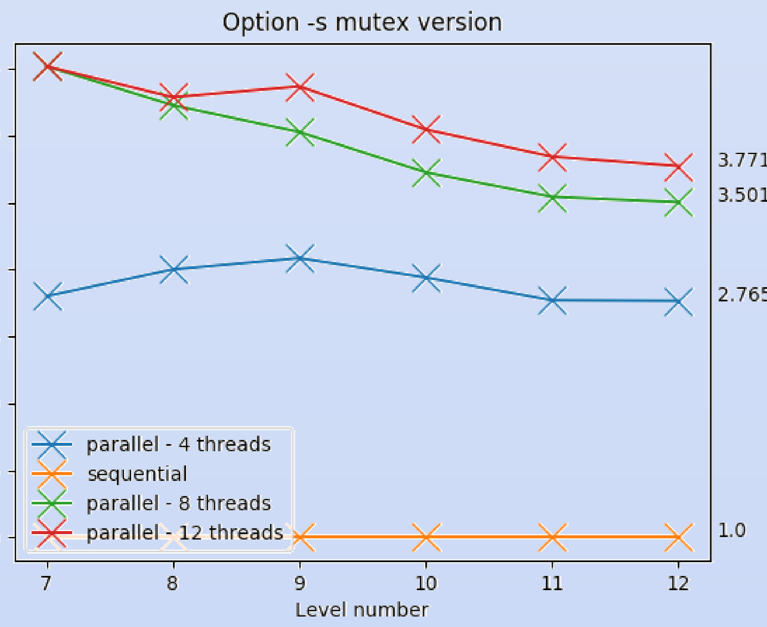
- Not possible to modify an element in a set (e.g. in the set of QuadEdges) due the use of hashtable for sorting. => Use of *mutable* attributes

Results

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

Refinement restricted to SURFACE Quadrants
At the end :
106,819 points, 320,520 QuadEdges and 55,533 Quadrants



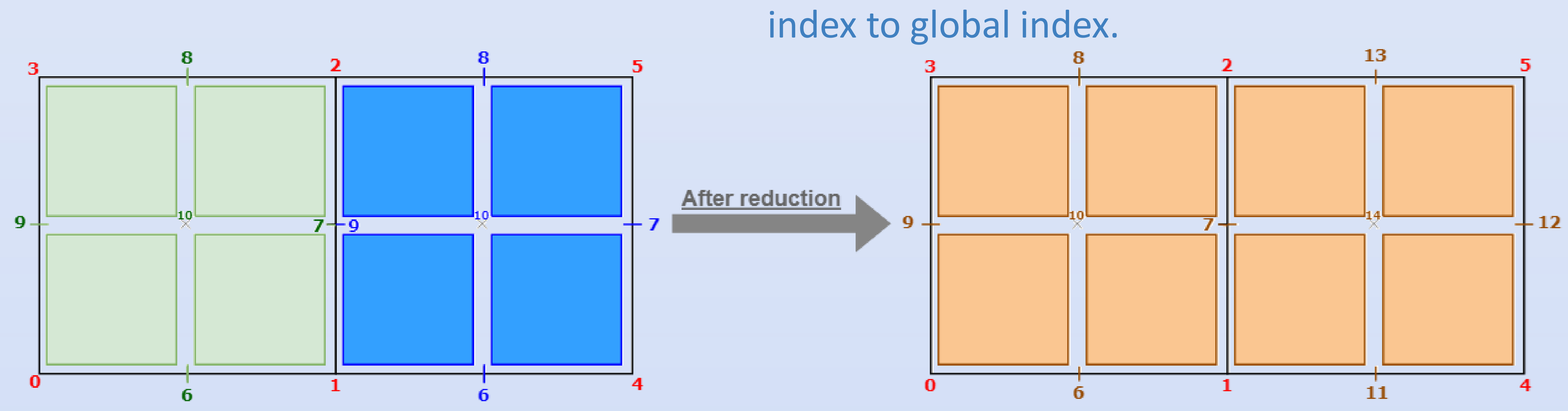


The mutex version speeds the -a option up to 3,5 times faster, and up to 4 times faster for the -s option. An increase in the number of threads speed up the generation. However the gain decreases because of the exchange and lock due to the mutex.

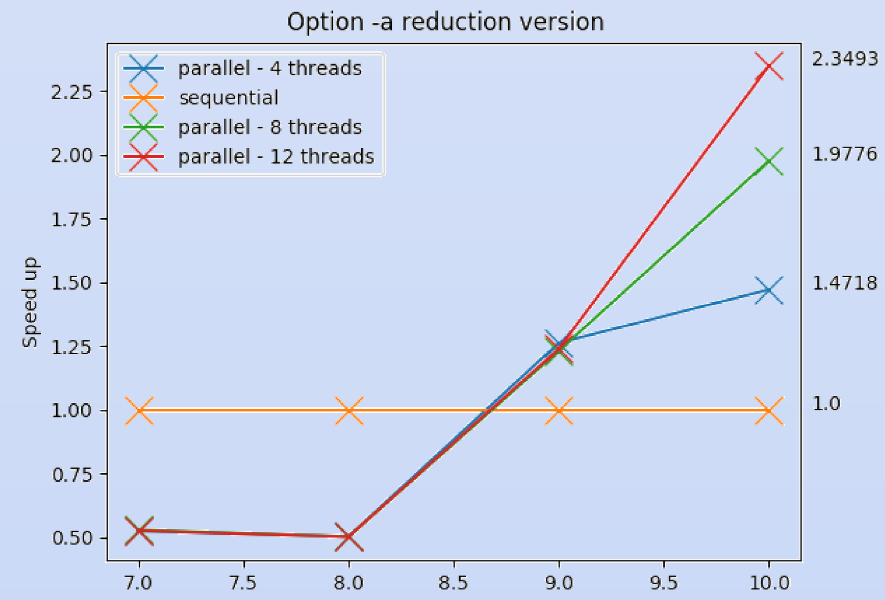
Reduction version

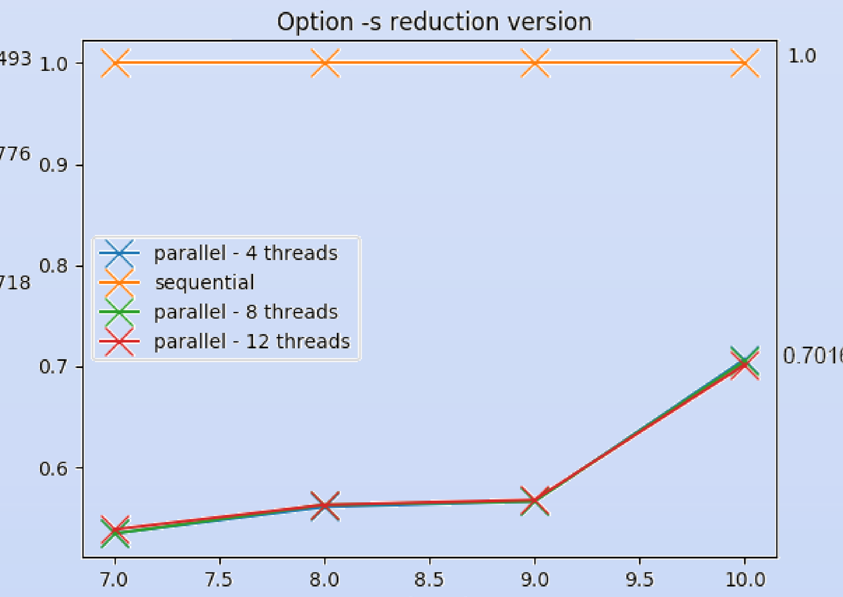
- No more critical regions.
- All threads have their own copy of Edges, Points and Quadrants filled in the accumulation part.
- The reduction part is done by a single thread and creates the final Edges, Points and Quadrants.
- Detection of identical points and update the local index to global index.

```
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)





The reduction version speeds the generation process up to 2,3 times faster, but only for the -a option, after refinement level 8. This version is efficient when there are enough quadrants (> 200,000) because of the slow sequential part of the reduction.

Conclusion

- **The mutex version is efficient especially when there are few quadrants**
- **The reduction version should be used only when there are a lot of quadrants : high level of refinement (> 9) and/or option for all quadrants (-a)**
- **Implementations avoiding critical sections produce a lot of new code, and it makes the whole project less maintainable**
- **Parallelism on this process is far from being trivial !**

Perspectives

- **Build a quadtree structure to represent all the Quadrants**
- **The reduction part is still performed sequentially and would require optimizations : a better algorithm for the reduction and/or the use of parallelism**
- **Make the reduction process only when the desired refinement level is reached**