**Algorithm:** PAROVER2

- **Objective:** Efficiently compute the area of every face of 3D meshes.
- **Supports:** Implemented using a functional paradigm with Chunks and Threads (C++).
- **Challenges:**
  - Computing and storing components in parallel.
  - Non-empty intersections: unknown in advance.
  - How many components each output has: unknown in advance.
  - Computing intersection areas of overlaid 2D meshes.

**Power of local topological formulae:**

1. **Area of a face from either:**
   - Set of vertex-edge incidences.
   - Vertex positions and neighborhood, for example.
   - Restricting edge slopes: 1 or -1.
   - Each vertex: sign bit (neighborhood).
   - Area: \( \sum s_i = 0 \cdot 4 + 9 \cdot 4 + 1 \cdot 4 \).

**Outface area computation:**

1. **Performed with a map-reduce:**
2. **Process input and compute each output vertex** with all its vertex-edge adjacencies. Two types:
   - Adjacency of one of the input maps.
   - Adjacency generated by an intersection of a pair of edges from the two input maps.
3. **Outface areas are accumulated.**

**Algorithm:** output adjacencies that are intersections of two input edges.

1. Input adjacency: \( h \), w.l.o.g., \( h \in M \).
2. \( h \) is adjacent to two faces \( f_1, f_2 \).
3. Vertex of \( h \): \( v \) in faces \( f_1, f_2 \).
4. Normal vector of \( h \) may need to be negated.
5. Area component (outface-id, area) computed for each outface.
6. Total area of each outface obtained by summing its components.

**Point location:**

1. **Input:** query point \( q \) from \( M \).
2. Create grid cell \( c \) containing \( q \).
3. Find the edge \( e \) in \( M \) that intersects a vertical ray from \( q \) at the closest point.
4. Otherwise, \( q \) is in one of the two faces adjacent to \( e \).

**Adjectives that are intersections:**

1. **Computing the adjacencies generated by the intersection of two input edges:** differs in two ways from previous case.
2. **Point location is not necessary (can be determined by the intersection).**

**Performance experiments:**

- **Test data:** overlapping square meshes.
- **Dual 14-core 2.0 GHz Xenon, 256 GB of RAM**
- **NVIDIA’s Thrust + OpenMP backend**
- **Parallel speedup of 6.3x (Turbo Boost reduces the speedup)**

**Conclusions and future work:**

- **Simple fast algorithm for computing the area of intersections.**
- **High-level functional programming style:** easily (?) portable code.
- **Extension to 3D:** volume of intersecting polyhedra.

**Acknowledgements:**

- We’ve been developing other software with similar ideas. E.g., 3D-EPU-Overlay, Union3, PinMesh.

**Oriented edges:**

- \( \{ \langle x, y, x, y, f, f \rangle \} \).
- \( f \) and \( f' \) are the ids of the two faces bounded by the edge.
- **Supports:**
  - Multiple components
  - Nested components
  - Self intersections

**Data representation:**

- “Soup” of edges.
- Oriented edges: \( \{ \langle x, y, x, y, f, f \rangle \} \).
- \( f \) and \( f' \) are the ids of the two faces bounded by the edge.
- **Supports:**
  - Multiple components
  - Nested components
  - Self intersections

**Novelties:**

- Expected linear execution time
- Grid indexing: efficient parallel uniform grid
- Computation with local information.
- Parallel: for multi-core computers
- Simple flat data structures: good for GPUs.
- Implemented using a functional paradigm with NVIDIA’s Thrust library.
- High-level implementation

**Storing area components:**

1. Size of the vector: \( 4x \) the number of input edges in the two meshes combined.
2. The \( h \)-th input edge will create output pairs numbered \( 4 \cdot h \) to \( 4 \cdot h + 3 \) — no need for synchronizations.
3. Vector sorted by outface-id and reduced-by-key.

**(slowest step: sort – no better alternative found yet)**