Lower Envelope Algorithms in $\mathbb{R}^3$

C++ Implementation, Parallelization and Benchmarking

Gianmarco Picarella @ Utrecht University

12-12-2023

Outline

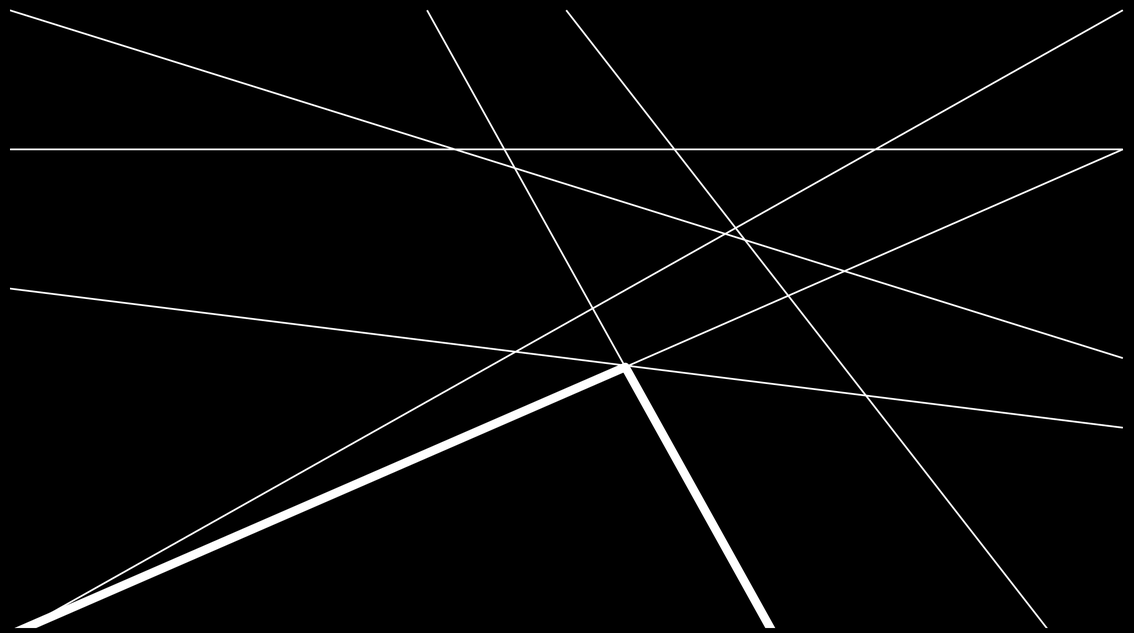
- Introduction
- Motivation and Goal
- Overview
- Algorithms
- Implementation Details
- Visualisation
- Benchmarks and Results

Introduction

Introduction (1)

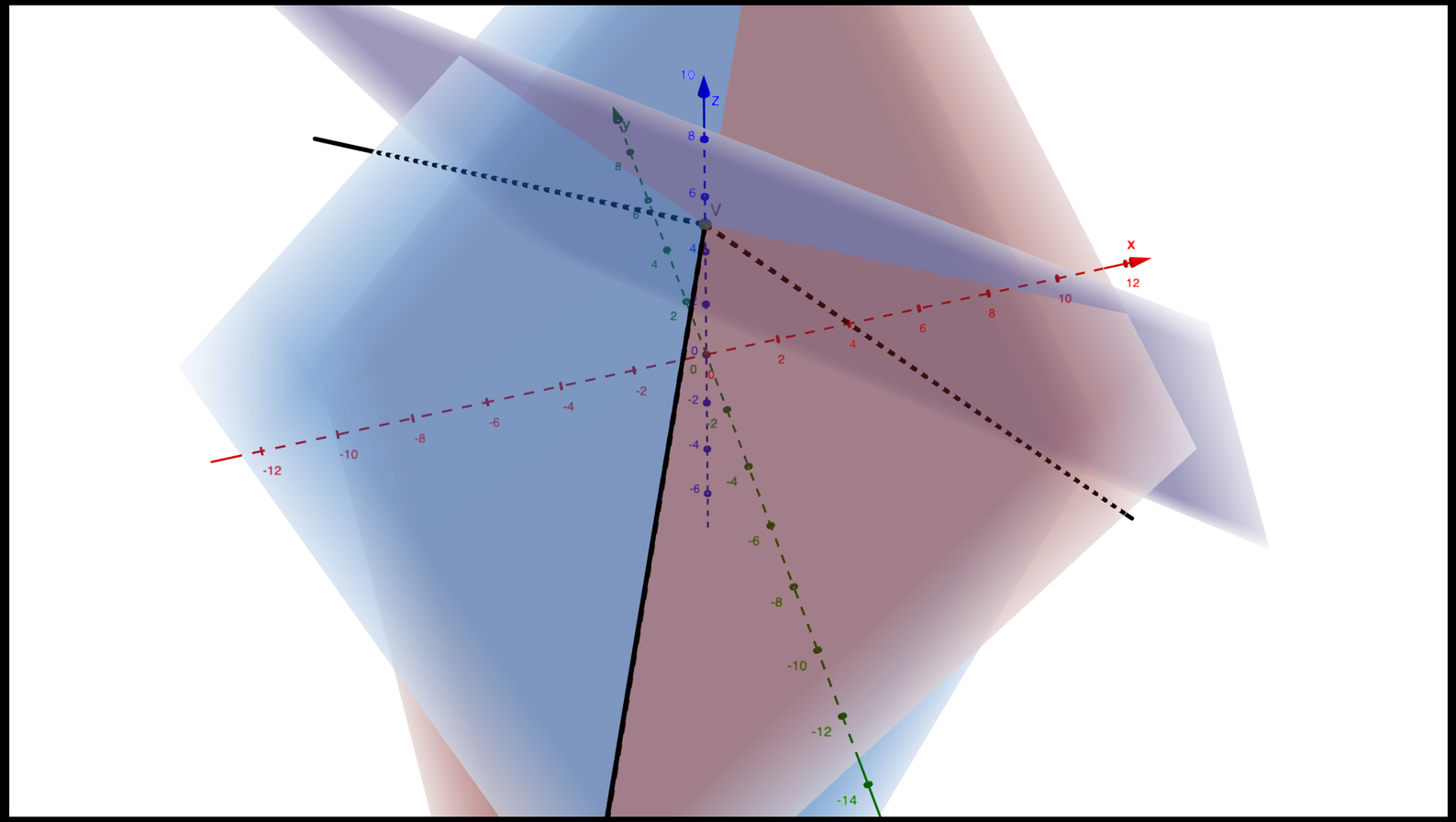
Lower Envelope L_0

- The function whose value at every point is the minimum of the values of the functions in the given set H



Introduction (2)

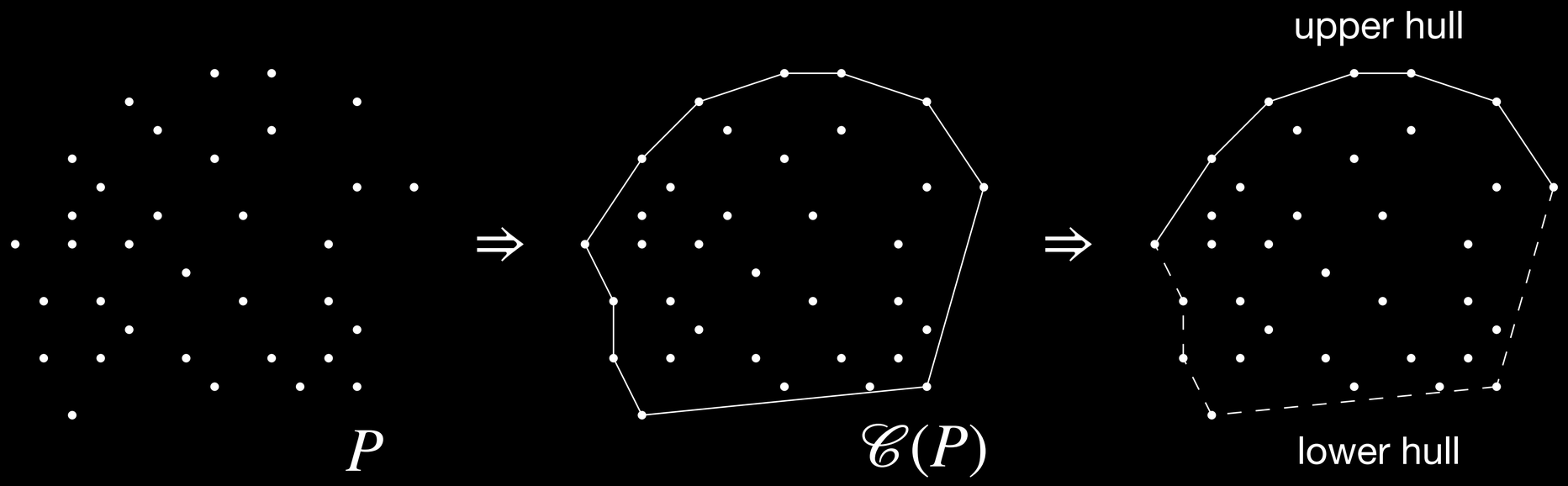
Lower Envelope L_0 (GeoGebra visualisation)



Introduction (3)

Convex Hull $\mathscr{C}$

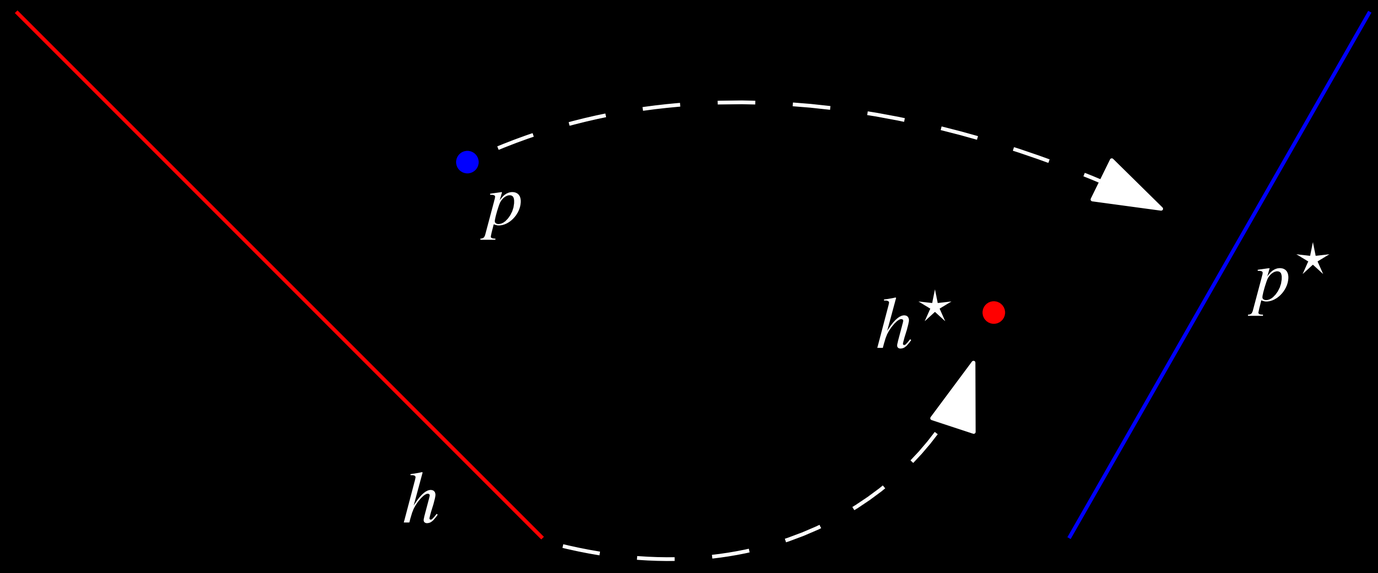
- The smallest convex polygon enclosing all the points in the input set
- Convex hulls can be partitioned in upper and lower hulls



Introduction (4)

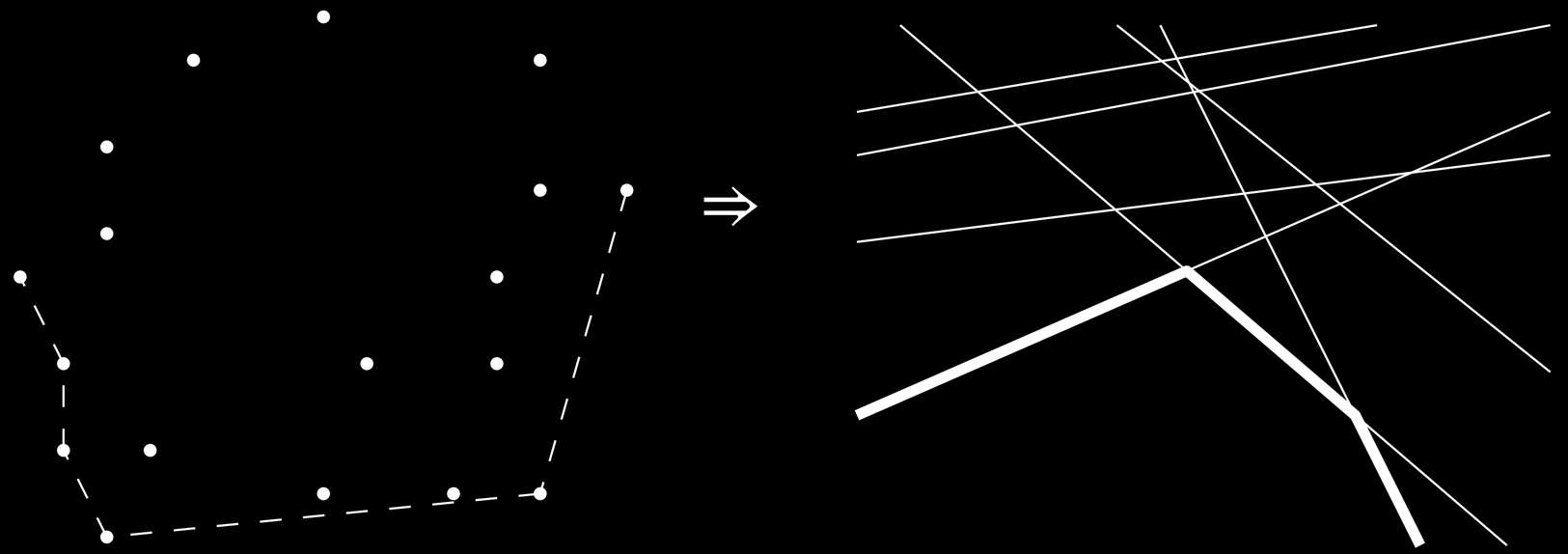
Duality Transform $(\cdot)^\star$

- Transformation from points to hyper-planes and vice-versa



Introduction (5)
Dual Relationship

- The lower hull in the primal is the lower envelope in the dual space



Introduction (6)

Background

- Frank Staals outlines in [2] an efficient parallel algorithm for computing 3D lower envelopes
- Partially based on T. Chan & E. Chen work [1]

Motivation and Goal

Motivation & Goal (1)

Motivation

- Convex hulls are fundamental to many applications
- Parallel algorithms for computing lower envelopes unlock parallel algorithms for the convex hull problem
- Modern CPUs provide n-fold speedups for embarrassingly-parallel problems
- Unclear if the algorithms and theoretical complexity in [2] are practically achievable

Motivation & Goal (2)

Goal

Exploit parallel computing in the context of 3D Lower Envelope algorithms

Motivation & Goal (3)

Questions

- Are the parallel algorithms in [2] practically feasible?
- What is an efficient way to implement such algorithms?
- How do the algorithms' running times relate to their theoretical complexities?
  A. As the input size is increased
  B. As the amount of parallel processors used is increased

Overview

Overview (1)

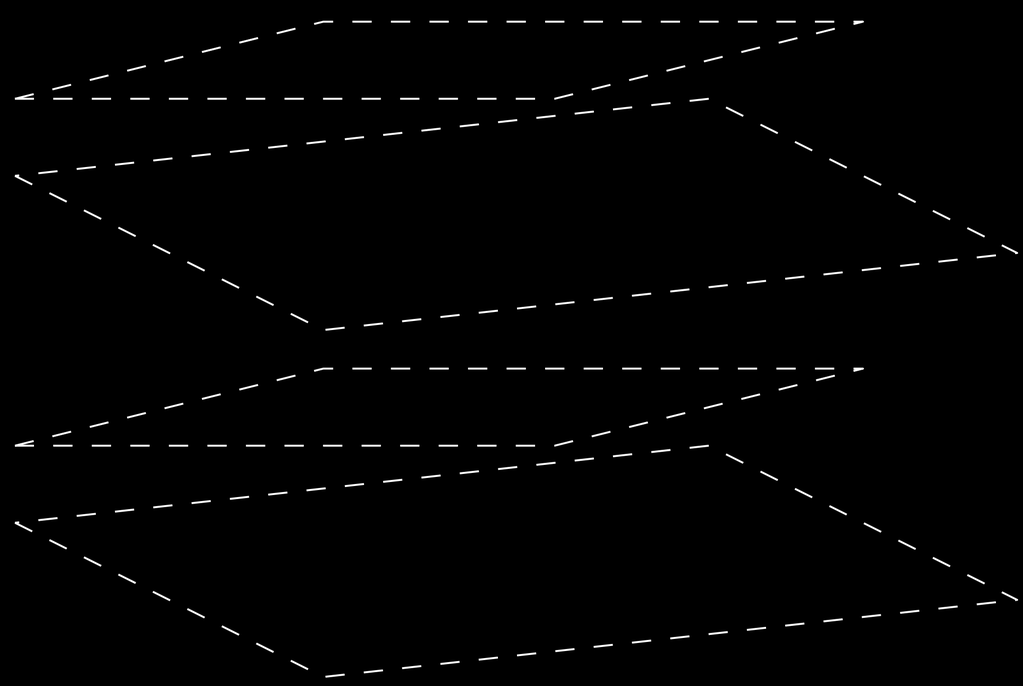
Efficient L_0 computation (main algorithm)

- Efficient divide and conquer, randomised algorithm outlined by F. Staals in [2]
- Given n input planes and p processors, the algorithm runs in $O\left(\frac{n}{p}\log n\right)$

Overview (2)

Efficient L_0 computation (main algorithm)

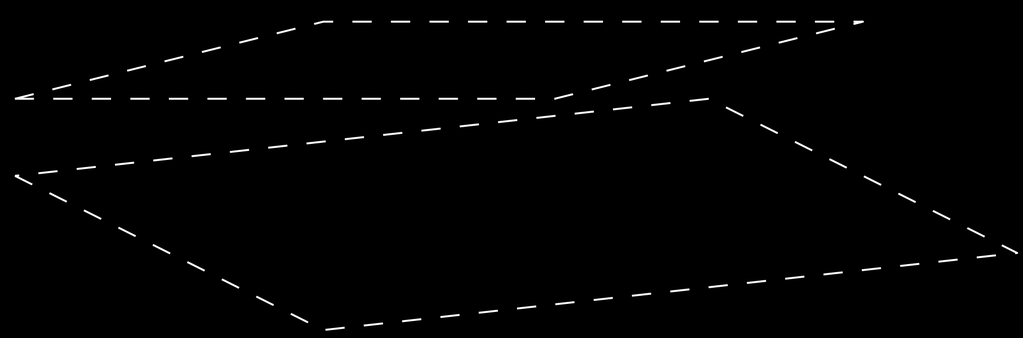
- Given a set of n planes H, compute $L_0(H)$



Overview (2)

Efficient L_0 computation (main algorithm)

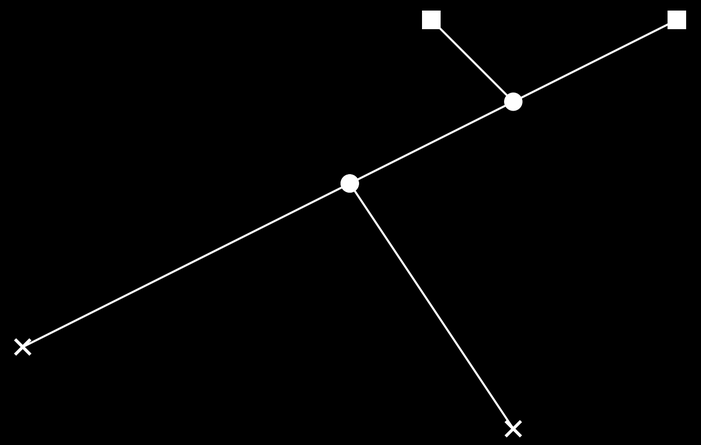
- Given a set of n planes H, compute $L_0(H)$
- Pick sample $S \subset H$ using double sampling scheme described in [2]



Overview (2)

Efficient L_0 computation (main algorithm)

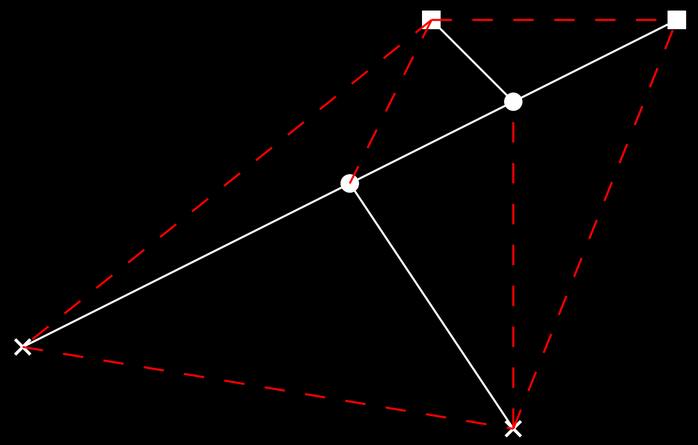
- Given a set of n planes H, compute $L_0(H)$
- Pick sample $S \subset H$ using double sampling scheme described in [2]
- Compute $L_0(S)$ and triangulate it



Overview (2)

Efficient L_0 computation (main algorithm)

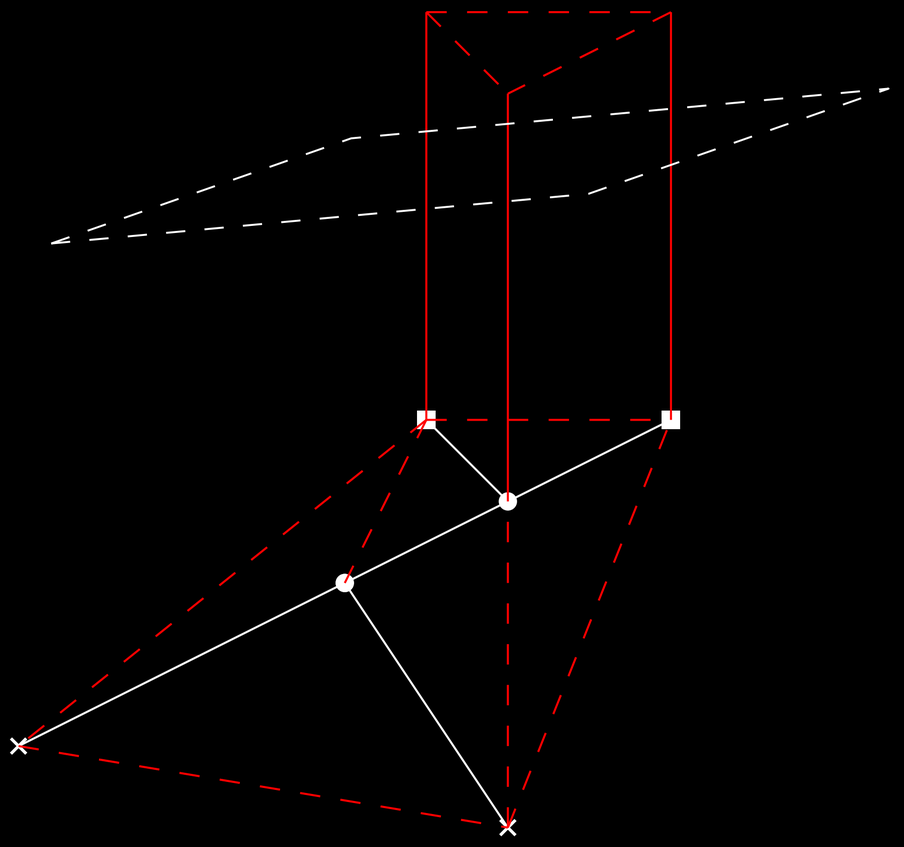
- Given a set of n planes H, compute $L_0(H)$
- Pick sample $S \subset H$ using double sampling scheme described in [2]
- Compute $L_0(S)$ and triangulate it



Overview (2)

Efficient L_0 computation (main algorithm)

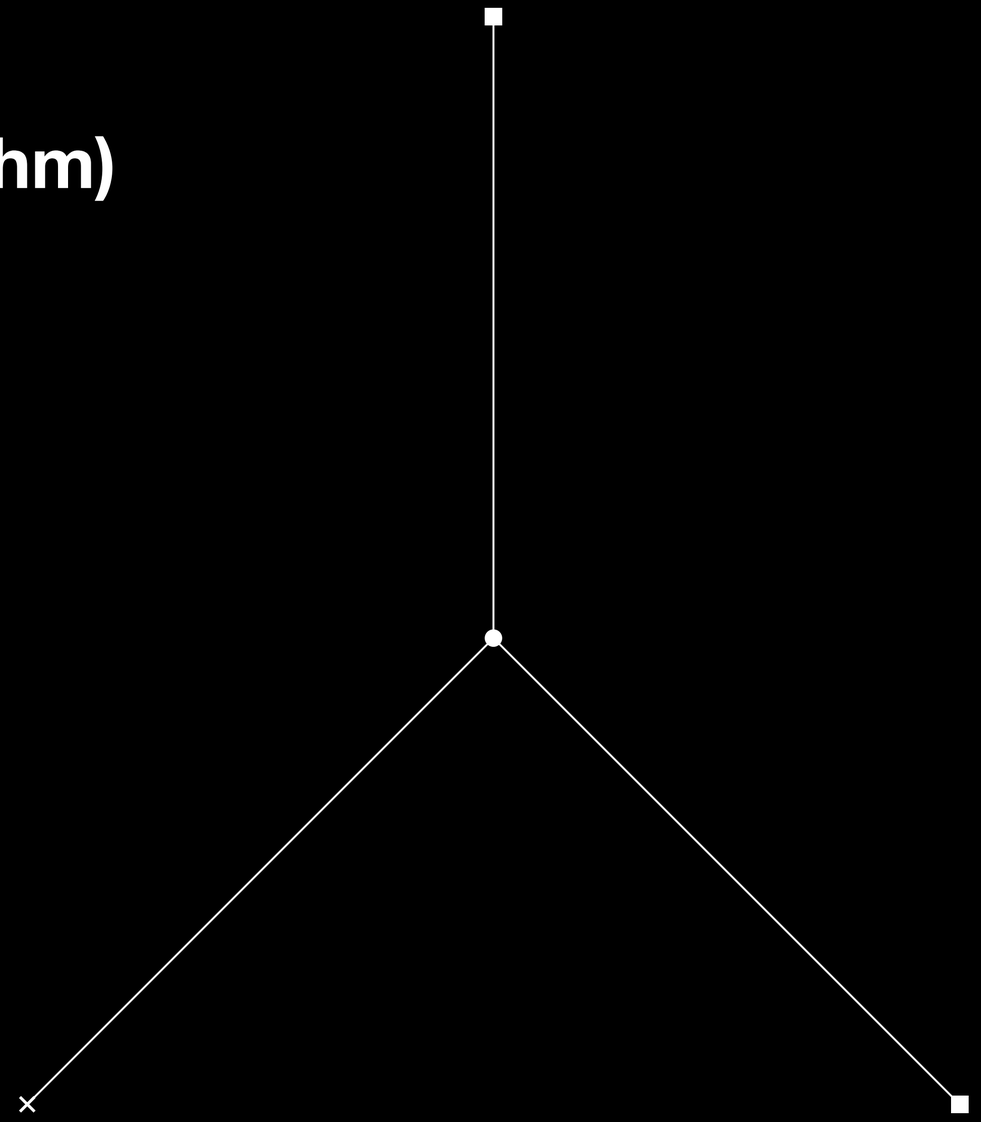
- Given a set of n planes H, compute $L_0(H)$
- Pick sample $S \subset H$ using double sampling scheme described in [2]
- Compute $L_0(S)$ and triangulate it
- Compute the conflict lists and solve all the sub-problems for each prism in $\Lambda_0(S)$



Overview (3)

Efficient L_0 computation (main algorithm)

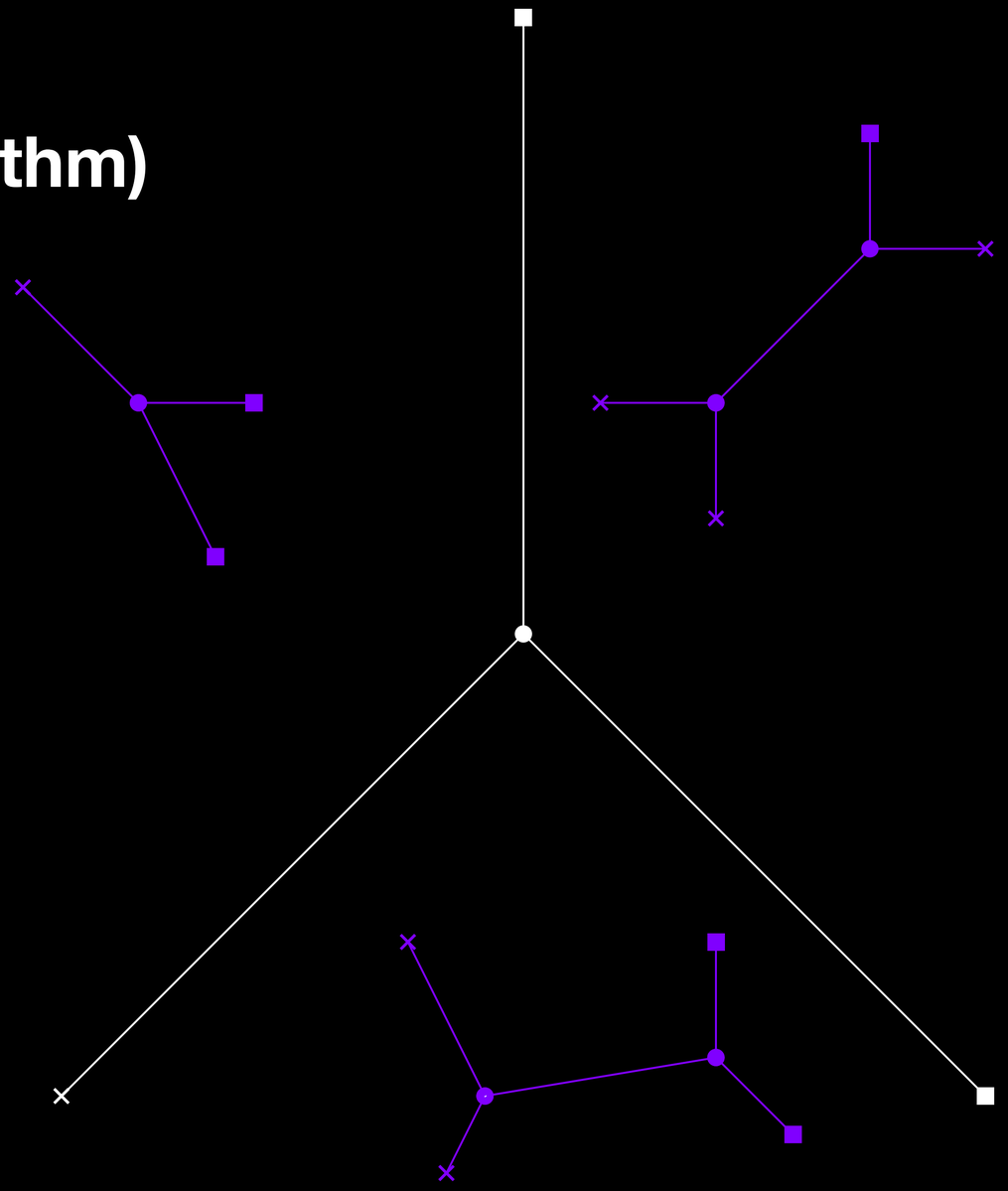
- Compute outer-most lower envelope



Overview (3)

Efficient L_0 computation (main algorithm)

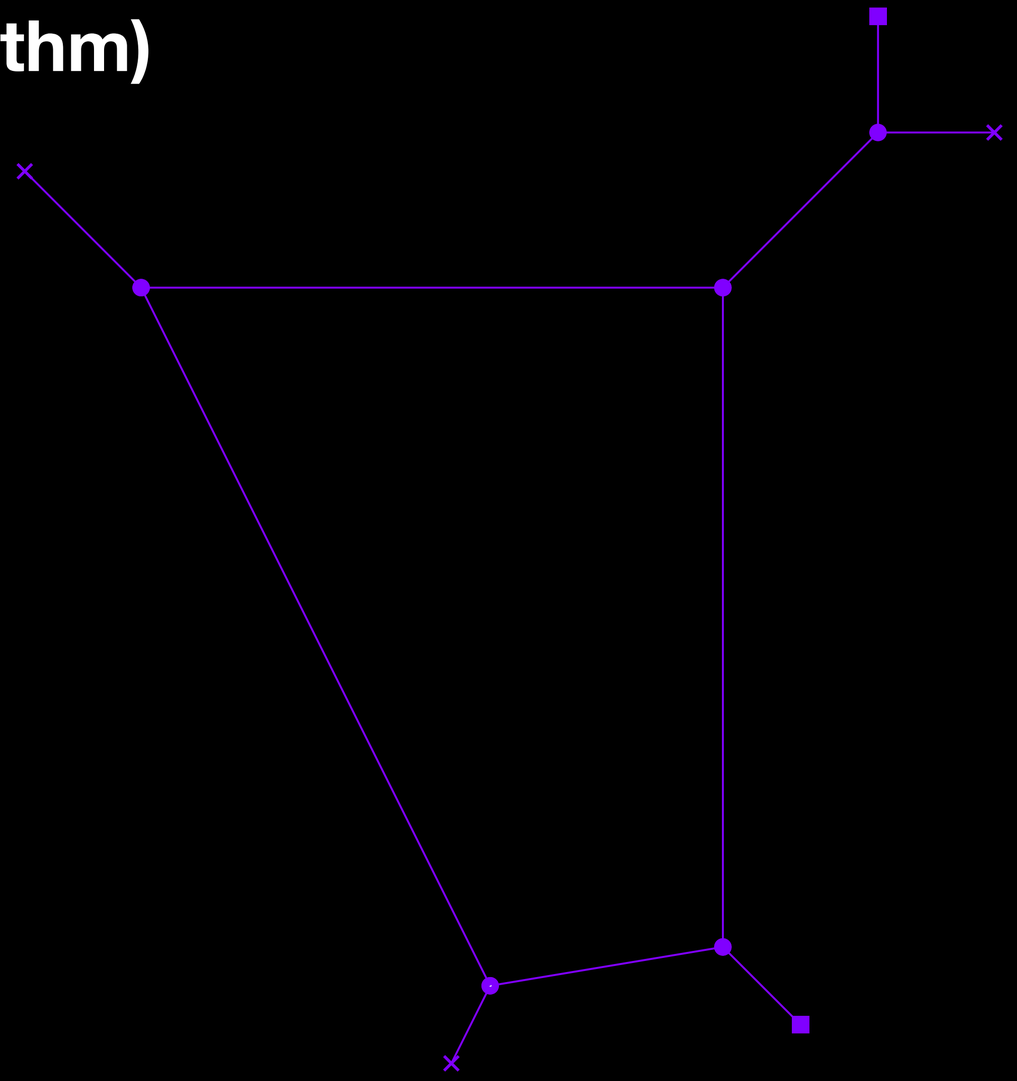
- Compute outer-most lower envelope
- For each prism, compute the conflict lists and find its lower envelope



Overview (3)

Efficient L_0 computation (main algorithm)

- Compute outer-most lower envelope
- For each prism, compute the conflict lists and find its lower envelope
- Merge adjacent lower envelopes along common edge



Overview (4)

Efficient L_0 computation (main algorithm)

- Various supporting algorithms are required:

  1. Lower envelope algorithm for small sized H
  2. Lower envelope triangulation algorithm
  3. Conflict lists algorithm
  4. Lower envelopes adjacency and merge algorithm

Overview (5)

Efficient L_0 computation (main algorithm)

- Focus of this presentation:

  1. Lower envelope algorithm for small sized H (brute force)

  2. Lower envelope triangulation algorithm

  3. Conflict lists algorithm (batch point location)

  4. Lower envelopes adjacency and merge algorithm

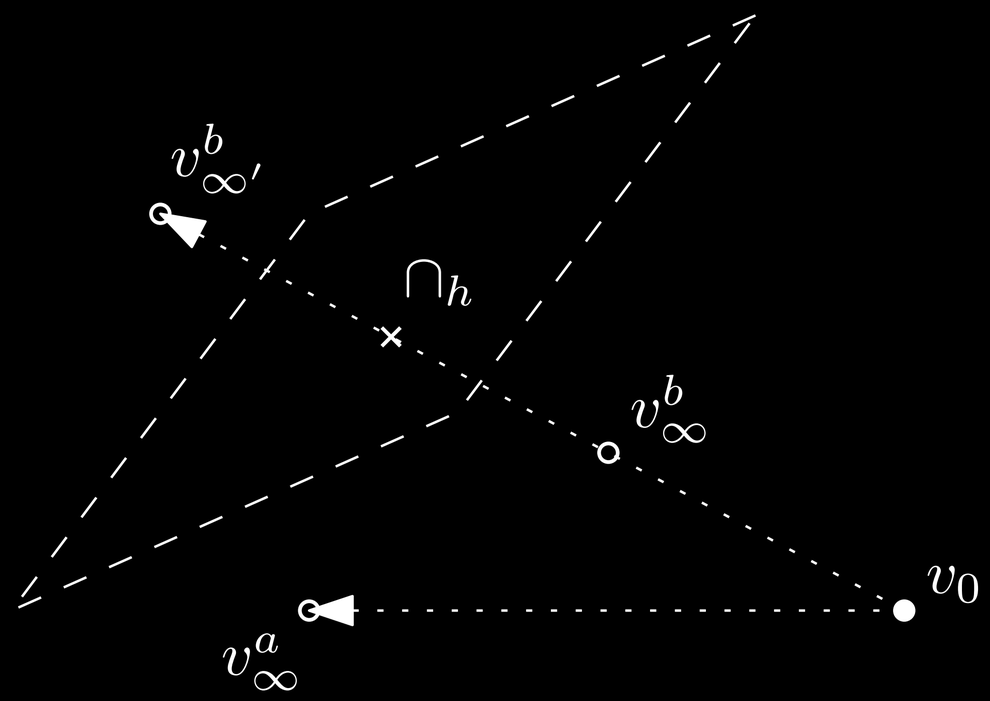
Overview (6)
Problem definitions

1. The Brute Force algorithm
   - Given a set of planes H, compute $L_0(H)$ using algorithms from [2]
3. The Batch Point Location algorithm
   - Given a set of planes H and a set of points Q, compute the z-sorted sequence of planes above each $q \in Q$

Overview (7)

Limitations

- Number of planes required for the double sampling scheme to work practically unfeasible
- Vertices at infinity may require special treatment
- Incomplete implementation

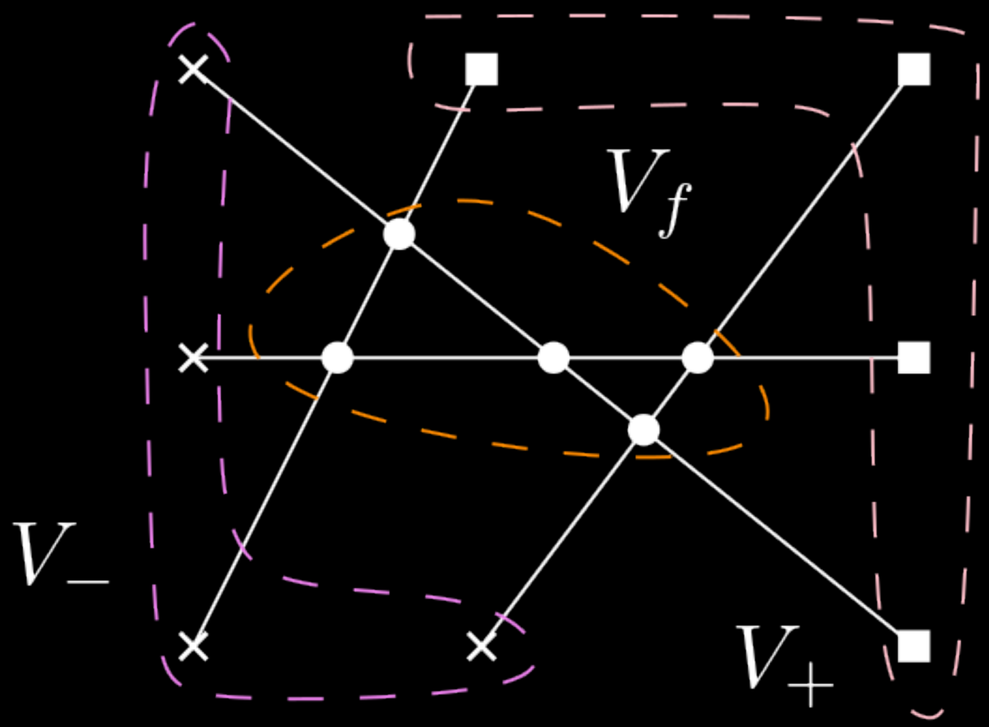


Algorithms

Algorithms (1)

Brute Force L_0

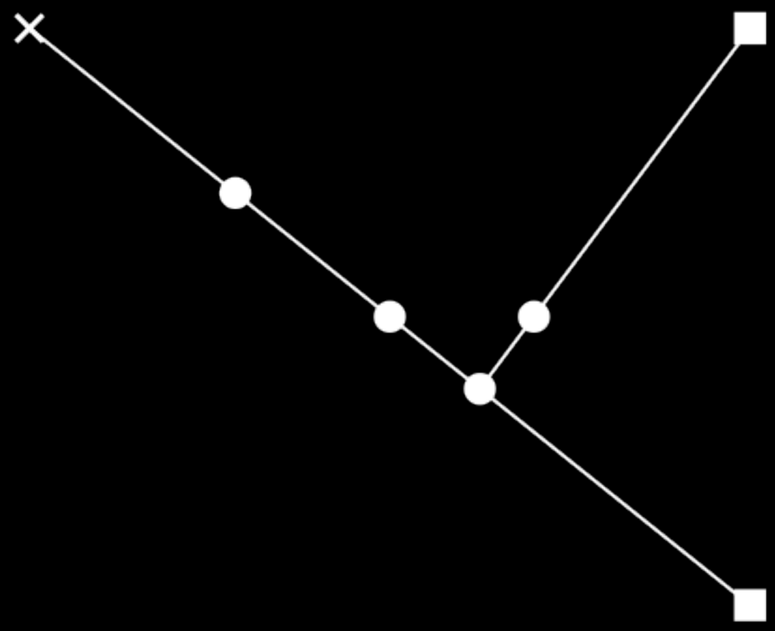
- Given a set of n planes H, compute $L_0(H)$
- Find the set of intersection lines generated by H
- Compute intersection and infinity vertices



Algorithms (1)

Brute Force L_0

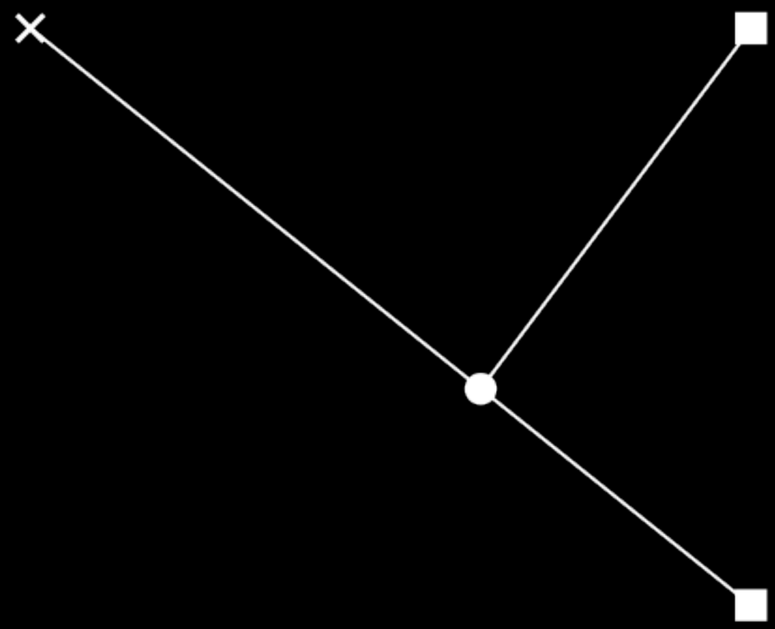
- Given a set of n planes H, compute $L_0(H)$
- Find the set of intersection lines generated by H
- Compute intersection and infinity vertices
- Filter out invalid segments



Algorithms (1)

Brute Force L_0

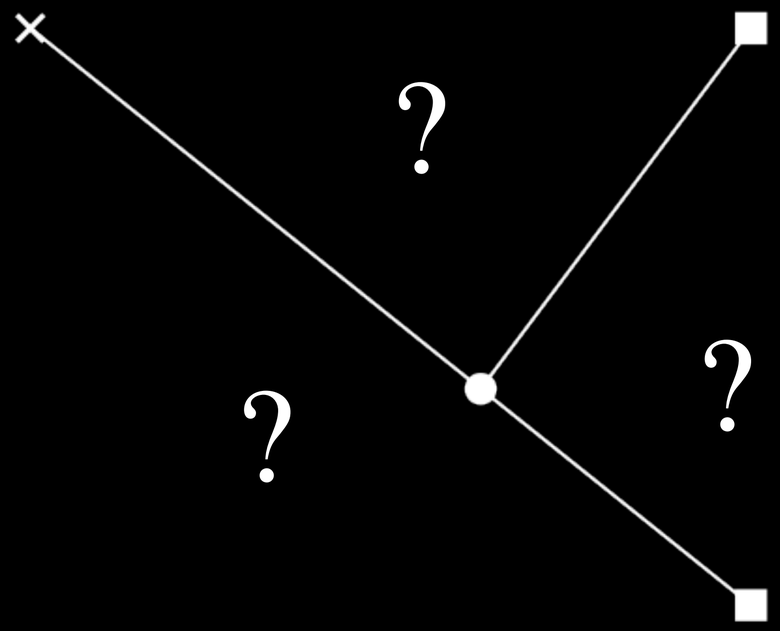
- Given a set of n planes H, compute $L_0(H)$
- Find the set of intersection lines generated by H
- Compute intersection and infinity vertices
- Filter out invalid segments
- Filter out invalid vertices



Algorithms (2)

Brute Force L_0

- For each face in $L_0(H)$, find the plane in H defining its surface



Algorithms (3)

Brute Force L_0

Algorithm 4: Compute $H_{\text{left}}(v_0, v_1)$

Data: $e = \overline{v_0 v_1} \in E$ and set $H' \subset H$ generating e

Result: $H_{\text{left}}(v_0, v_1)$

1 $Z \leftarrow (0, 0, 1)$;

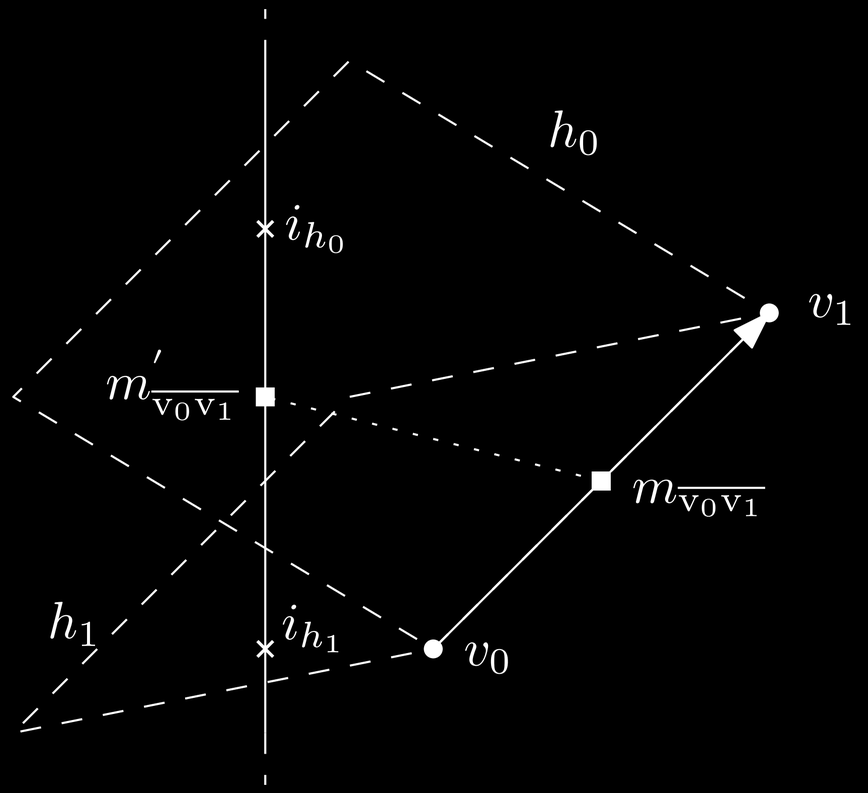
2 $T \leftarrow 1000$;

3 $m_{\overline{v_0 v_1}} \leftarrow \frac{v_0 + v_1}{2}$;

4 $m'_{\overline{v_0 v_1}} \leftarrow -(\hat{e} \times Z) \cdot T$;

5 $l \leftarrow$ Vertical line passing through $m'_{\overline{v_0 v_1}}$;

6 **return** $\arg\min_{h \in H'} (l \cap h)_z$



Algorithms (4)

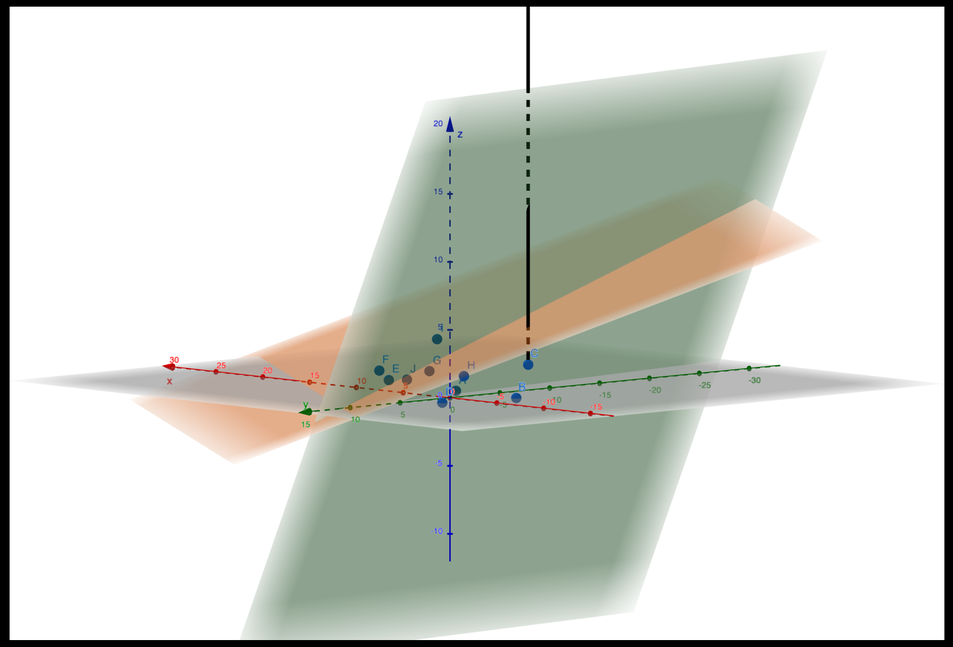
Brute Force L_0

- Given n input planes and p processors, the algorithm runs in $O\left(\frac{n^4}{p}\right)$

Algorithms (5)

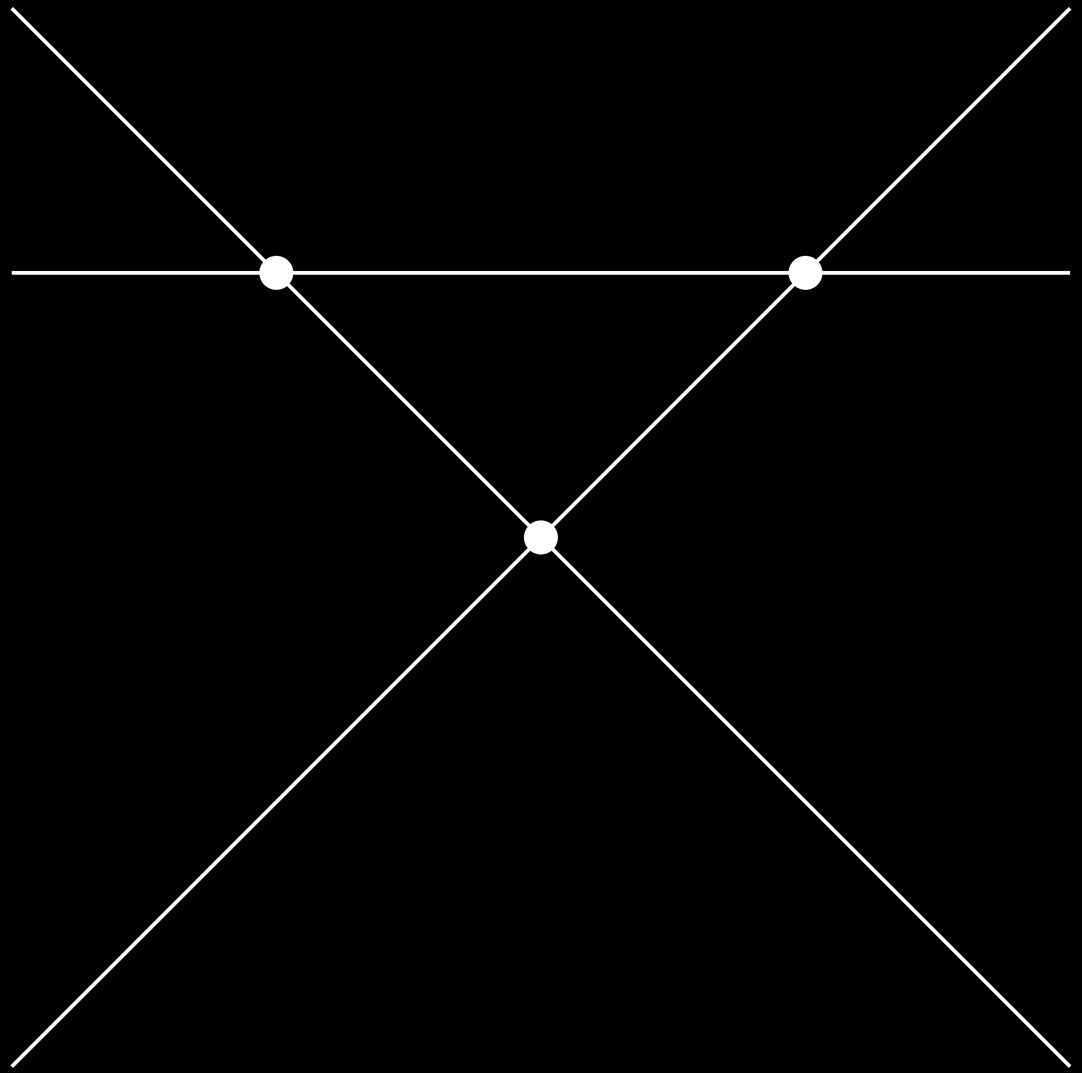
Batch Point Location (GeoGebra visualisation)

- Given a set of planes H and a set of points Q, compute the z-sorted sequence of planes $\forall q \in Q$



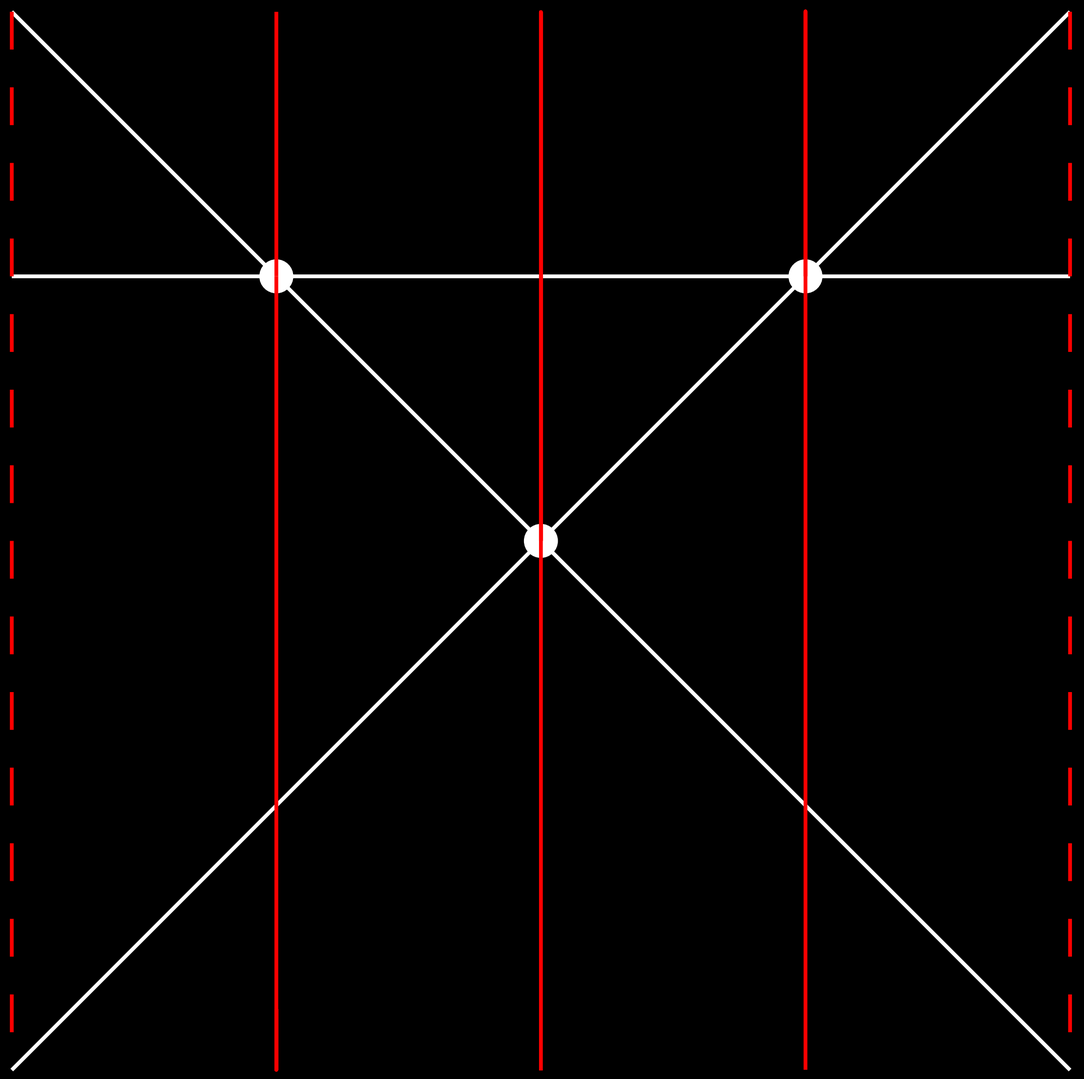
Algorithms (6)
Batch Point Location

- Partition $\mathbb{R}^2$ based on the projected intersection lines generated by H



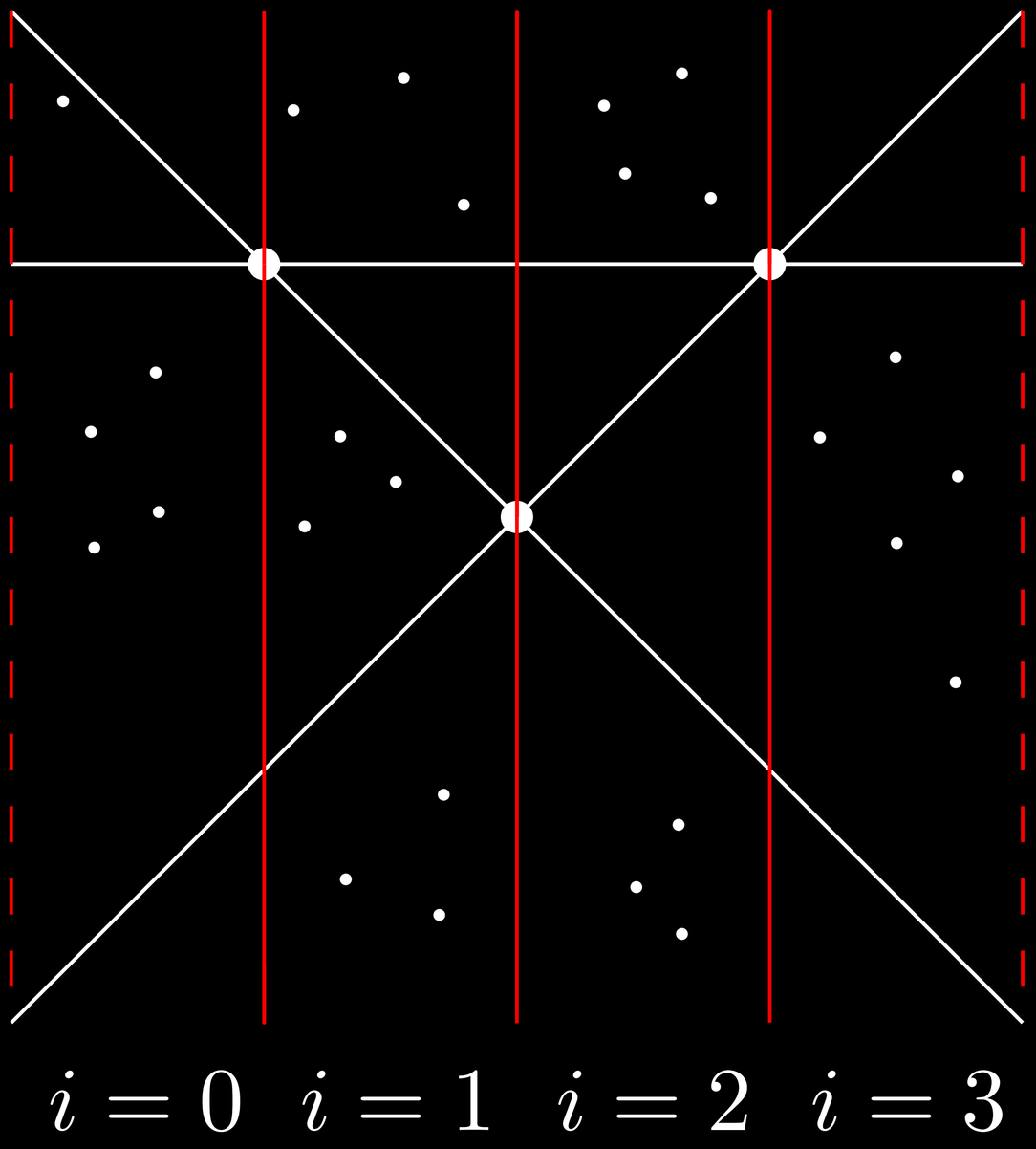
Algorithms (6)
Batch Point Location

- Partition $\mathbb{R}^2$ based on the projected intersection lines generated by H
- Further subdivide $\mathbb{R}^2$ with vertical slabs



Algorithms (6)
Batch Point Location

- Partition $\mathbb{R}^2$ based on the projected intersection lines generated by H
- Further subdivide $\mathbb{R}^2$ with vertical slabs (Binary search)
- Multiple points may share the same face (Caching)
- Many faces are not used at all (Lazy Computation)



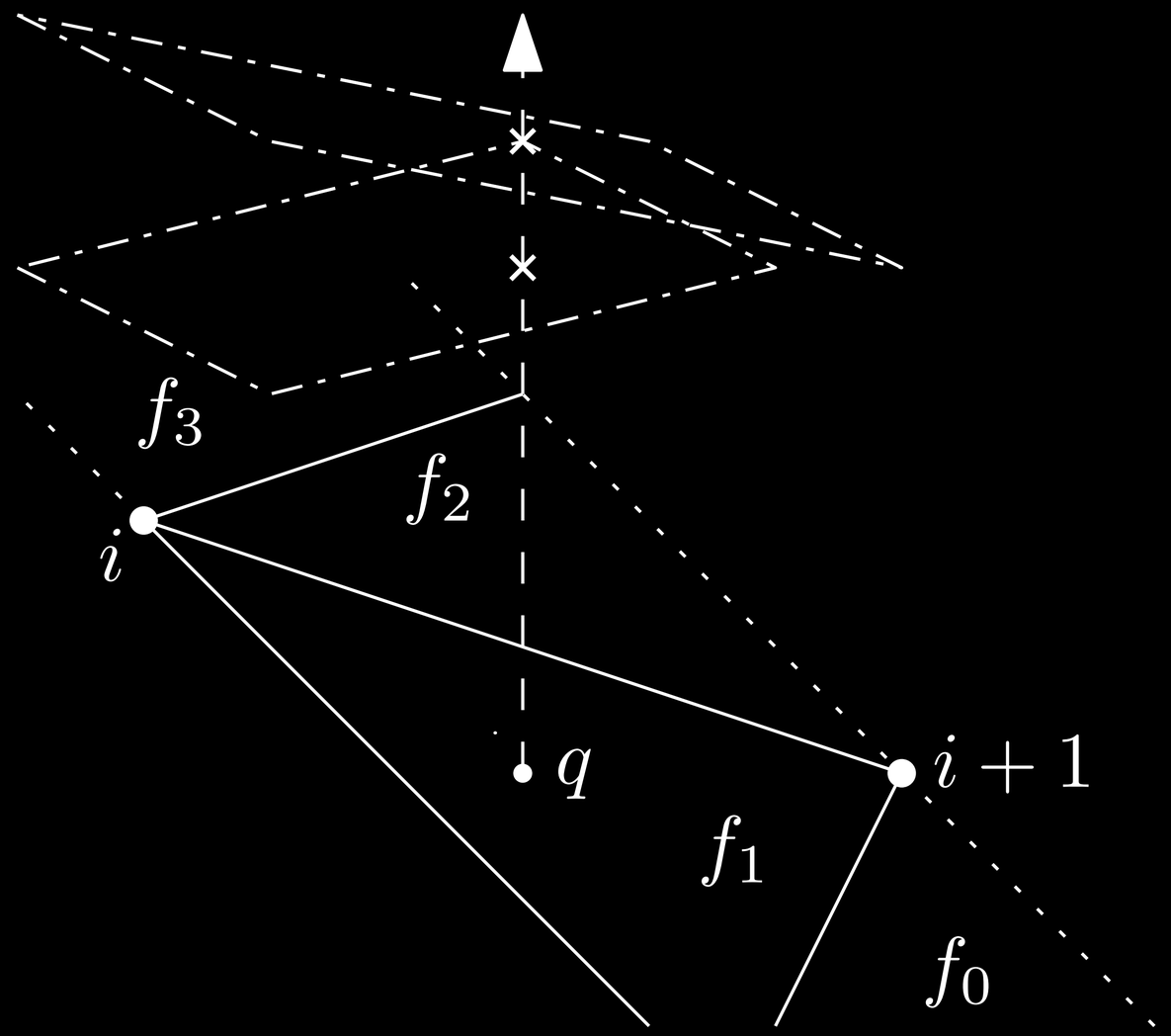
Algorithms (7)
Batch Point Location

- Divide computation into three independent phases:

  1. Slabs lookup

  2. Faces lookup

  3. Z-sorted planes computation



Algorithms (8)
Batch Point Location

- Given n input planes, k query points and p processors, the algorithm runs in

$$O\left(\frac{n^4 \log n + k \log n + on^2 \log n + fn \log n}{p}\right)$$

- With o = # slabs used and f = # faces used

Algorithms (9)
Batch Point Location

- Find the set of planes passing below each prism in $L_0(S)$ with $S \subset T$
- Small and bounded amount of prisms $\Lambda_0(S)$
- Many planes to test in $T - S$
- Achieve efficiency with duality:
  1. $Q = (T - S)^\star$ is a set of query points (expected to be very large)
  2. $H = \Lambda_0(S)^\star$ is a set of planes (expected to be very small)

Algorithms (10)
Missing algorithms

- Lower envelope triangulation, adjacency and merge algorithms

Algorithm 9: Assemble final envelope

Data: $P, L = L_0(C_i)$
Result: $L_0(H)$
$E_0 \leftarrow (v_0^e, v_1^e, \triangle_{\text{idx}})\ \forall \triangle \in L_0(S)\ \forall e \in \triangle$;
$L_0 \leftarrow L$;
$T_0 \leftarrow |P|$ booleans set to false;
while $\exists a, b \in E_i \mid a_2 \neq b_2$ **do**
  Lexicographically sort E_i;
  Remove edges defined by only one triangle;
  foreach *Pair of consecutive edges* $(a, b) \in E_{i-1}$ **do**
    if $T[a] \vee T[b]$ **then**
      continue;
    end
    $T[a], T[b] \leftarrow$ true, true;
    $m \leftarrow$ Merge envelopes (a, b)using Merge$_\triangle$;
    Store envelope inL_i;
    $E_i \leftarrow$ Insert $(a_0, a_i, \min(a_{idx}, b_{idx}) \forall a \in E_{i-1}$;
  end
  Set all entries in T to false;
end

Algorithm 11: Triangulates each face of $L_0(H)$

Data: $L_0(H)$
Result: A triangulated $L_0(H)$
$s, p, E \leftarrow 0, -1, \prec_\triangle L_0(H)$;
for $i \leftarrow 1$ **to** $|E|$ **do**
  if $i = |E| \vee E[s]_{plane} \neq E[i]_{plane}$ **then**
    for $k \leftarrow s + 1$ **to** i **do**
      if $ps > -1 \wedge (k = p \vee E[k]_{end} = E[p]_{start})$ **then**
        continue;
      end
      $E \leftarrow$ Insert edge $\langle E[s]_{\text{start}}, E[k]_{\text{end}} \rangle$;
      $E \leftarrow$ Insert edge $\langle E[k]_{\text{end}}, E[s]_{\text{start}} \rangle$;
    end
    $s, p \leftarrow i, -1$;
  end
  else if $E[i]_{end} = E[s]_{start}$ **then**
    $ps \leftarrow i$;
  end
end
return $\prec E$;

Implementation Details

Implementation Details (1)

Development Environment and 3rd Party Libraries

- Using CMake and Git
- Using C++ 17 with Visual Studio 2022 (Windows)
- Using CGAL for exact arithmetic
- Using SFML for visualisation
- Using Catch2 for Test-Driven Development
- Using OneAPI Intel TBB for C++ code parallelisation

Implementation Details (2)

Pair-wise Intersections

- Reduced from n^2 to $\approx \frac{n^2}{2}$ intersection tests
- Work divided in tiles based on the amount of instructions
- Contiguous tiles scheduled to the same thread
- Results stored locally to each thread and spliced at the end

Implementation Details (3)

Pair-wise Intersections

```
class Observer : public tbb::task_scheduler_observer
{
    size_t& myFinalSize;
    std::vector<Point2>& myOutData;

public:
    Observer(std::vector<Point2>& someOutData, size_t& aFinalOutCount)
        : tbb::task_scheduler_observer(), myOutData(someOutData), myFinalSize(aFinalOutCount)
    {
        observe(true);
    }

    ~Observer()
    {
        observe(false);
        for (const auto& chunk : myData)
        {
            myFinalSize += chunk.size();
        }
    }

    void on_scheduler_entry(bool /*worker*/)
    {
        CGAL_precondition(myData.local().empty());
    }

    void on_scheduler_exit(bool /*worker*/)
    {
        const int start = myCounter.fetch_add(myData.local().size());
        std::copy(myData.local().begin(), myData.local().end(), myOutData.begin() + start);
    }

    tbb::enumerable_thread_specific<std::vector<Point2>> myData;
    std::atomic<size_t> myCounter{ 0 };
};
```

```
{
    Observer obs(anOutData.myVertices, finalSize);
    static tbb::affinity_partitioner ap;
    tbb::parallel_for(tbb::blocked_range2d<int, int>(0, anOutData.myLines.size(), 0, anOutData.myLines.size()),
        [&](tbb::blocked_range2d<int, int>& r) {
            if (r.rows().end() > r.cols().begin())
            {
                for (int i = r.rows().begin(); i < r.rows().end(); ++i)
                {
                    for (int k = r.cols().begin(); k < r.cols().end(); ++k)
                    {
                        if (i <= k) continue;
                        const auto intersection = CGAL::intersection(anOutData.myLines[k], anOutData.myLines[i]);
                        if (intersection)
                        {
                            const Point2* point = boost::get<Point2>(&*intersection);
                            CGAL_precondition(point != nullptr);
                            obs.myData.local().emplace_back(*point);
                        }
                    }
                }
            }

        });

    const int start = obs.myCounter.fetch_add(obs.myData.local().size());
    std::copy(obs.myData.local().begin(), obs.myData.local().end(), anOutData.myVertices.begin() + start);
}
```

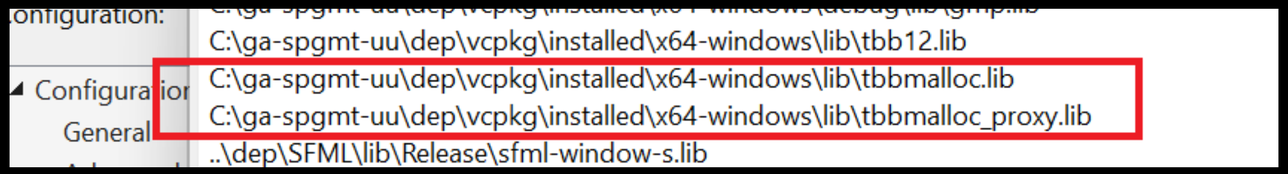
Implementation Details (4)
Allocation and Deallocation Overhead

- Around 35% of the execution time spent for allocation and deallocation
- Nested data structures are expensive to deallocate
- Solutions:
    1. Use TBB scalable and cache aligned allocator
    2. Use plain arrays instead of arrays of nested structures

Implementation Details (5)
Allocation & Deallocation Overhead

```cpp
// OLD
std::vector<std::tuple<FT, std::vector<FT>>> mySlabs;

// NEW
std::vector<Point2> myVertices;
```

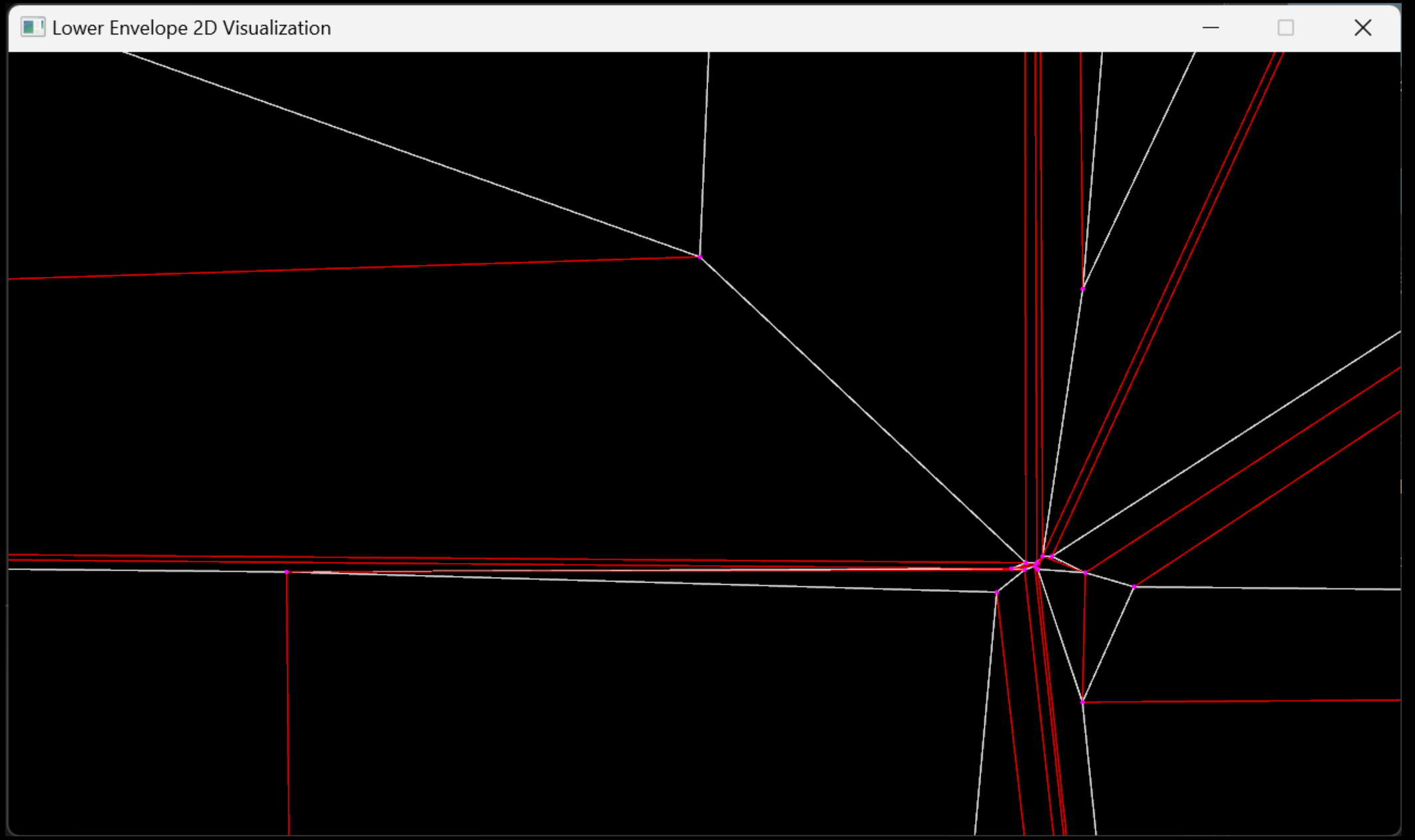


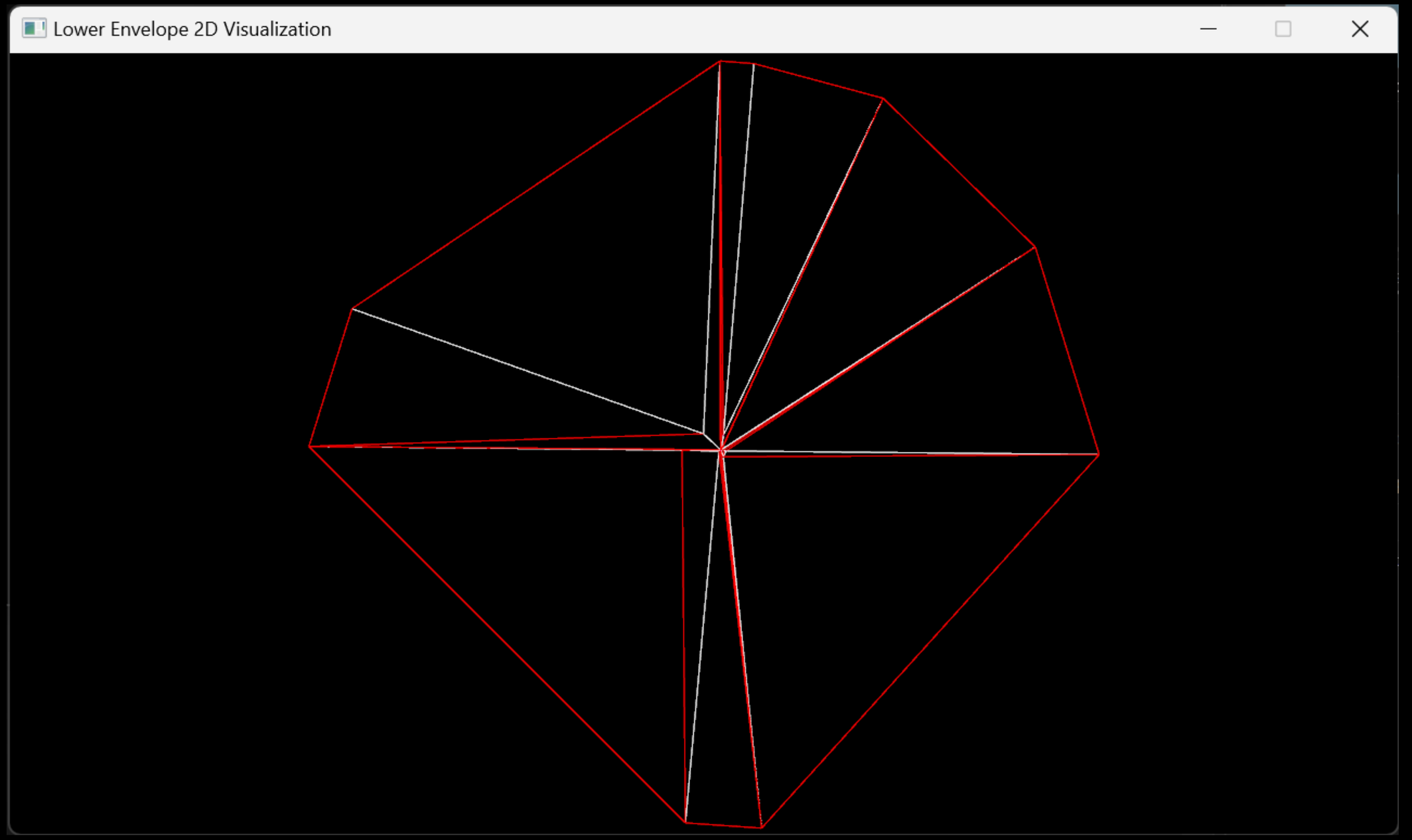

```
24
25    #include <oneapi/tbb/tbbmalloc_proxy.h>
26
```

Implementation Details (6)
Exact Arithmetic

- Using CGAL::Lazy_exact_nt
- Around 25% penalty per arithmetic operation
- Dynamic allocations
- Stored values impossible to inspect while debugging
- Doubles or floats would significantly improve performances, but inexact computations require special treatment

Visualisation


Lower Envelope 2D Visualization


Lower Envelope 2D Visualization

Benchmarks and Results

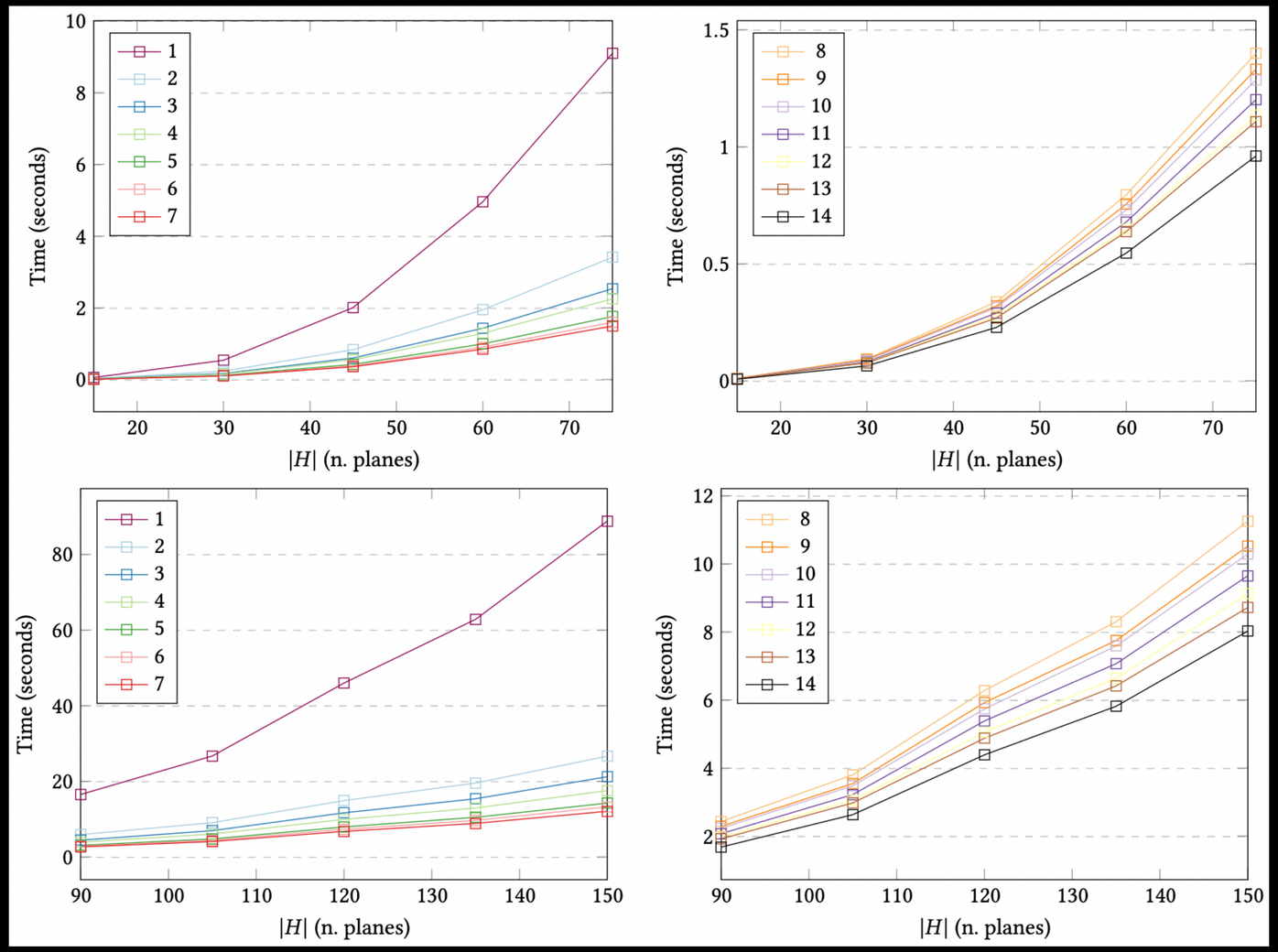
Benchmarks and Results (1)

Introduction

- Experiments run for 1 to 14 core versions
- CPU frequency scaling disabled
- 10 runs per input instance with SD $\approx$1% of total runtime

Benchmarks and Results (2)

Brute Force Algorithm



Benchmarks and Results (3)

Brute Force Algorithm

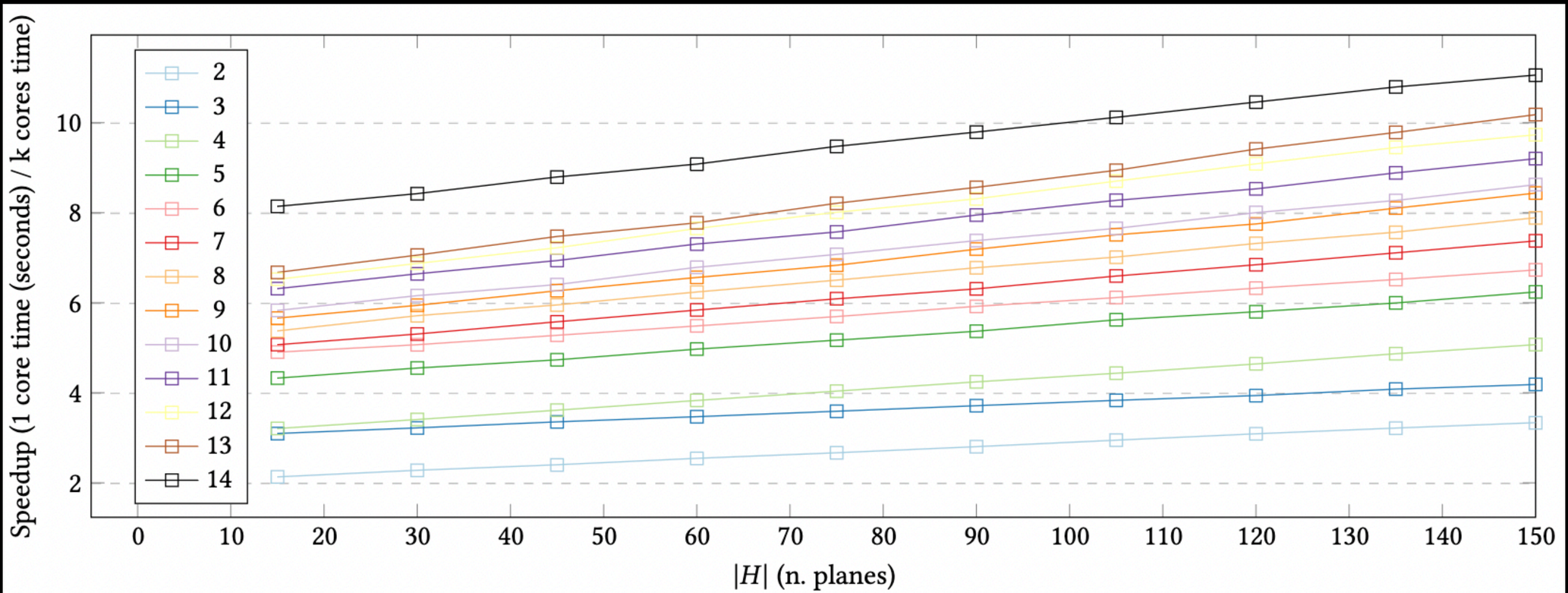


Figure 15: Brute-force algorithm parallel speed-ups

Benchmarks and Results (4)

Brute Force Algorithm

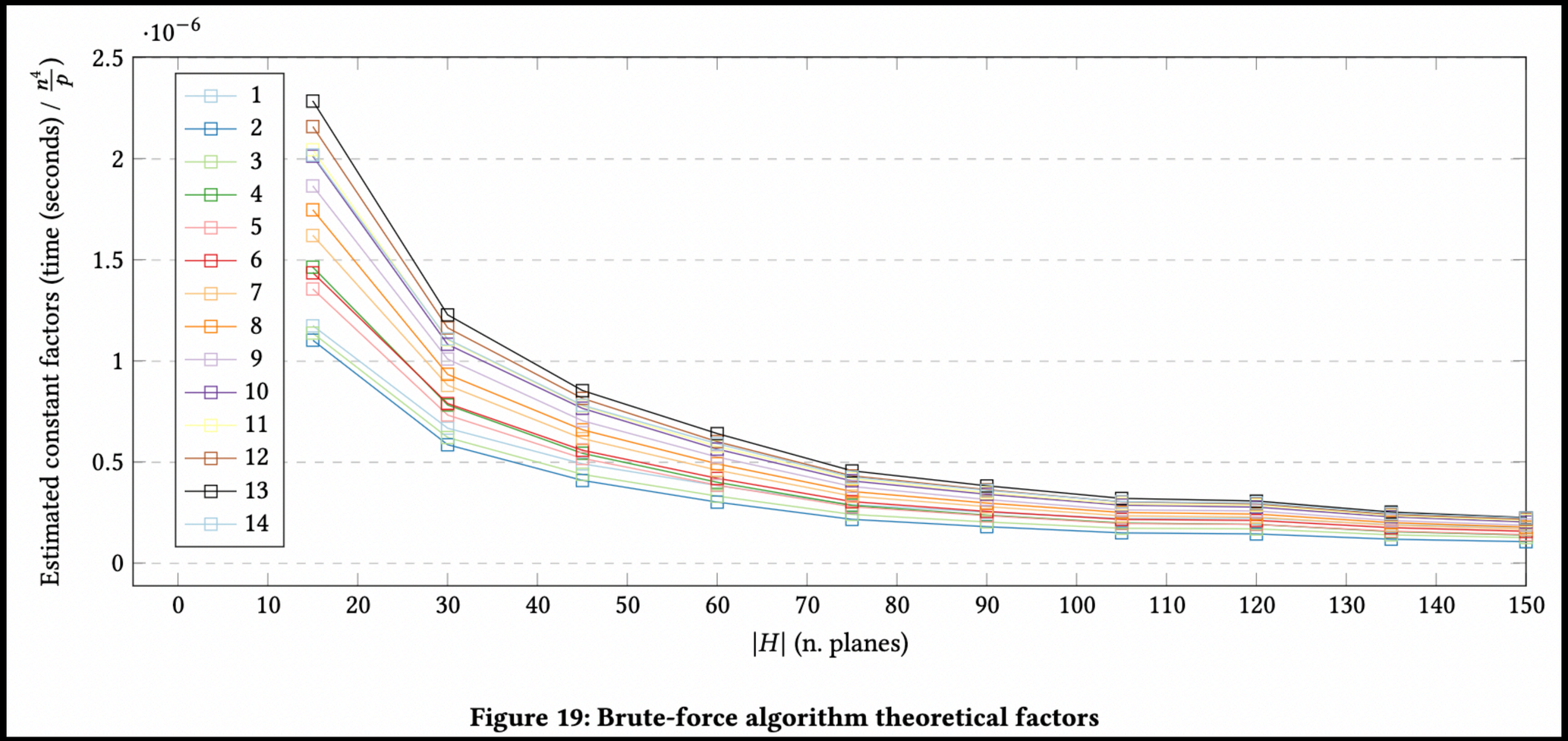
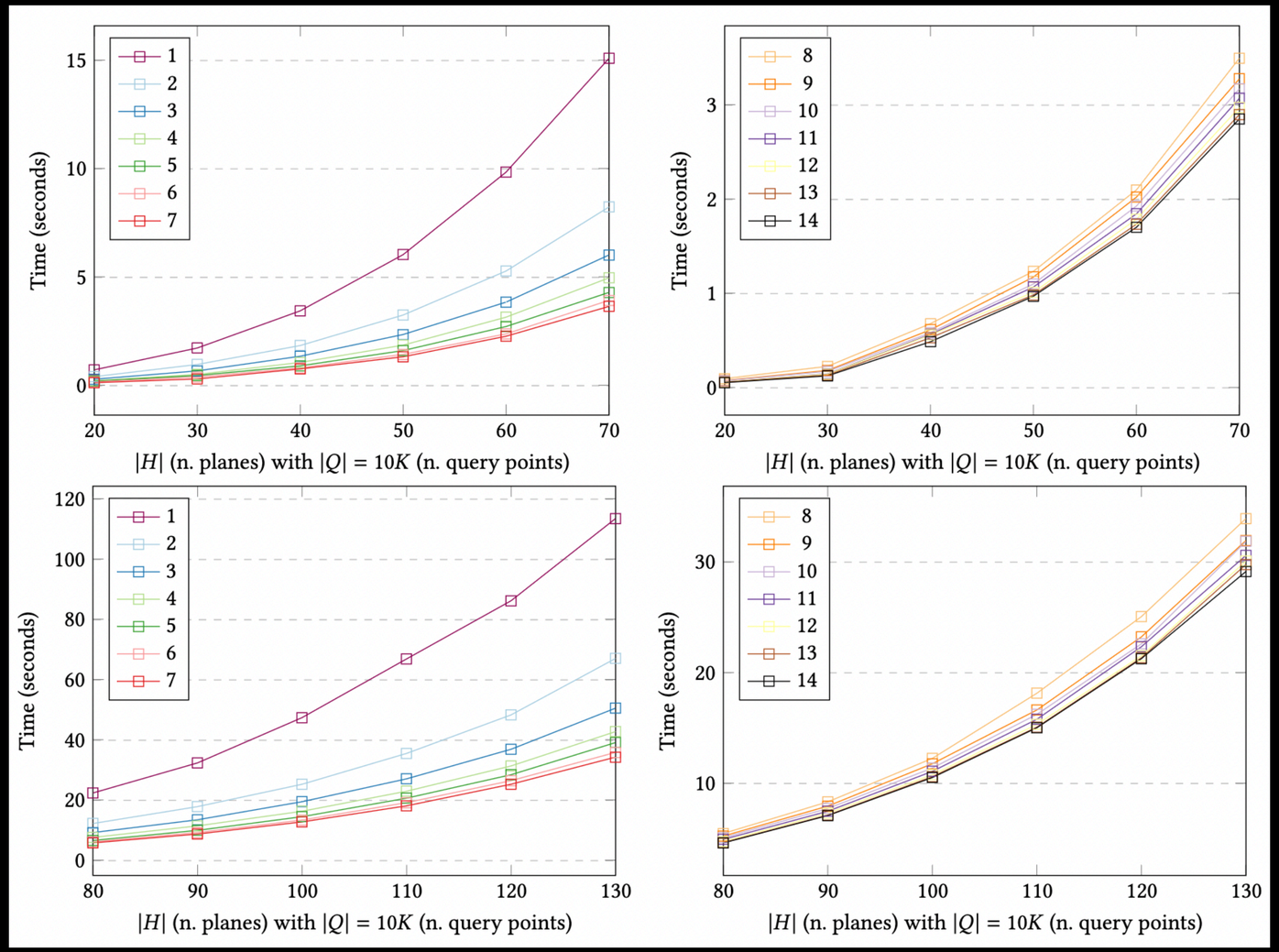


Figure 19: Brute-force algorithm theoretical factors

Benchmarks and Results (5)

Batch Point Location Algorithm



Benchmarks and Results (6)

Batch Point Location Algorithm

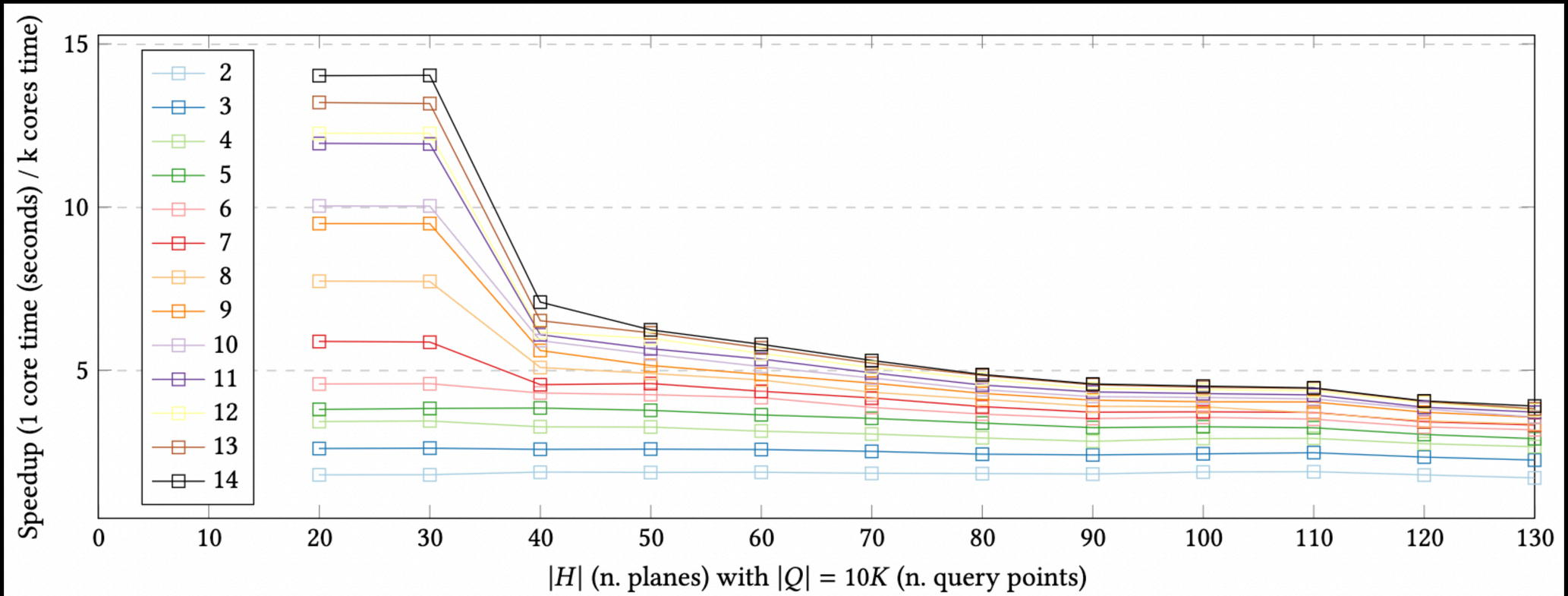


Figure 17: Batch-Point-Location algorithm parallel speed-ups

Benchmarks and Results (7)

Batch Point Location Algorithm

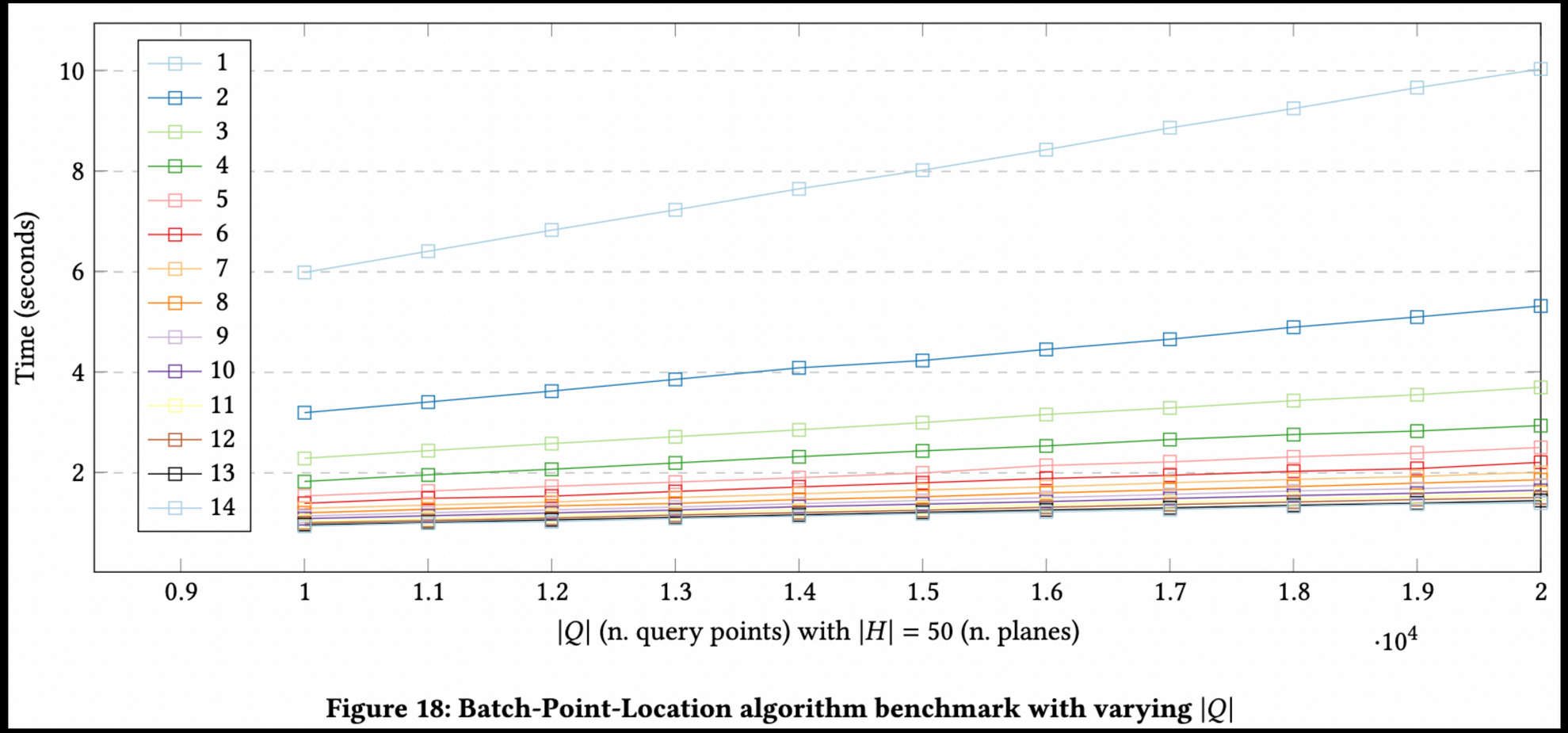


Figure 18: Batch-Point-Location algorithm benchmark with varying $|Q|$

Benchmarks and Results (8)

Batch Point Location Algorithm

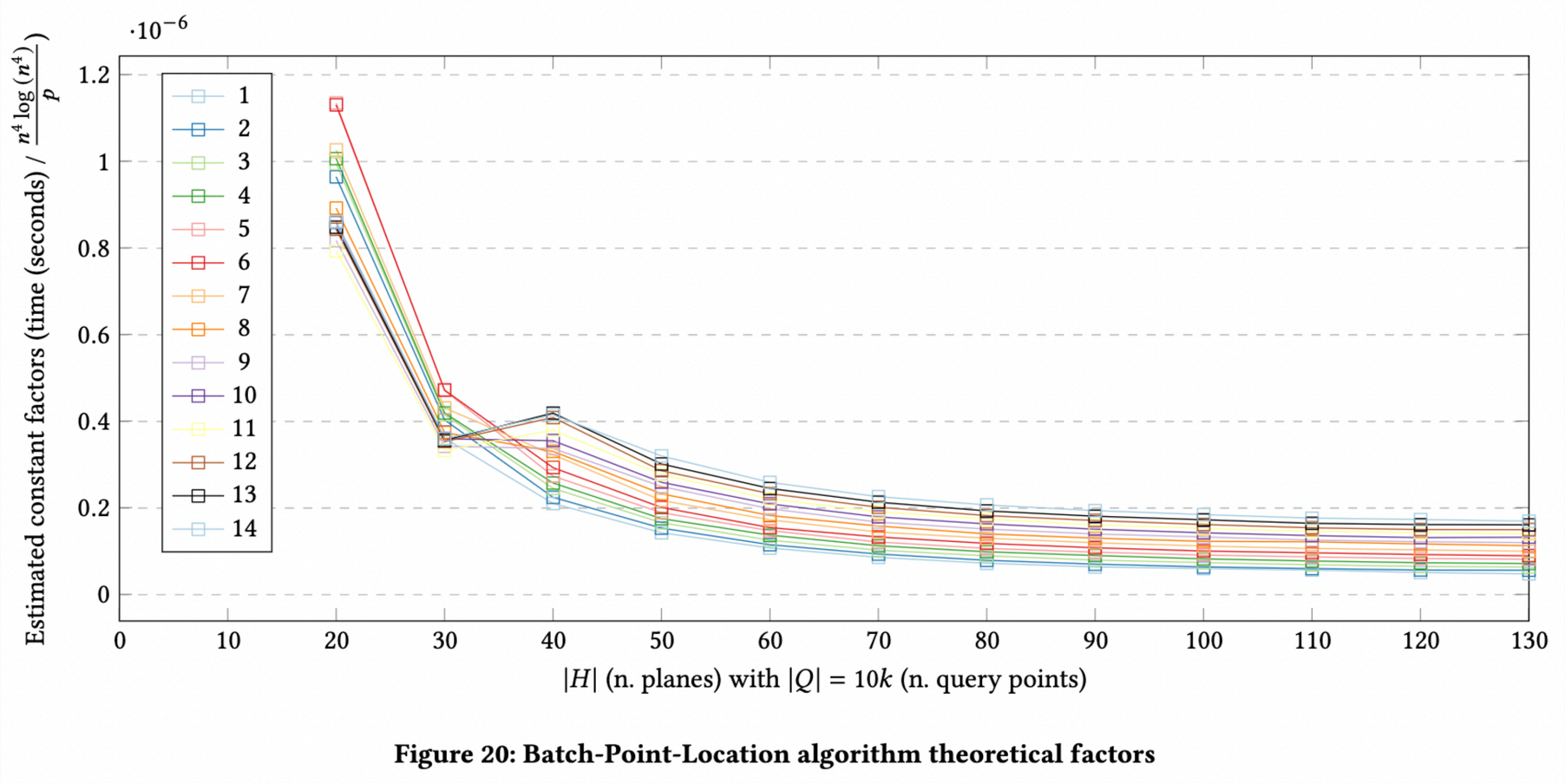


Figure 20: Batch-Point-Location algorithm theoretical factors

Benchmarks and Results (9)

Future Work

- The main algorithm requires additional theoretical work concerning the sampling scheme used
- The batch point location algorithm requires I/O and cache-efficient slab and zone search routines
- It would be interesting to adapt the current implementations to use inexact arithmetic and measure the performance improvement

References

1. Timothy Chan and Eric Chen. 2010. Optimal in-place and cache-oblivious algorithms for 3-d convex hulls and 2-d segment intersection. Computational Geometry 43 (10 2010), 636–646. https://doi.org/10.1016/j.comgeo.2010.04.005.
2. Frank Staals. 2023. Parallel 3D Lower Envelope/Convex Hull. Technical Report (2023).
3. Gianmarco Picarella. 2023. 3D Lower Envelope Algorithms. https://github.com/gianmarcopicarella/ga-spgmt-uu.

Thank you! Questions?