Lower Envelope Algorithms in $\mathbb{R}^3$: C++ Implementation, Parallelization and Benchmarking

Gianmarco Picarella
g.picarella@students.uu.nl
Universiteit Utrecht
Utrecht, Utrecht, The Netherlands

ABSTRACT

This report outlines the work done for the small GMT project (**INFOMSPGMT**) I worked on from September 5th to December 1st 2023, supervised by Frank Staals at Utrecht University. The project's main goal involves designing, implementing and benchmarking algorithms required for computing 3D envelopes in $\mathbb{R}^n$ based on prof. Staals' theoretical notes [4]. In particular, four algorithms are discussed: a brute-force, 3d lower envelope algorithm, a batch point location algorithm, a conflict lists algorithm and a lower envelope algorithm based on random sampling and divide-and-conquer techniques. Finally, a performance investigation is carried out to establish how each algorithm's runtime relates to the input size and number of processors used.

CCS CONCEPTS

• **Theory of computation → Randomness, geometry and discrete structures**; **Parallel algorithms**; **Divide and conquer**.

KEYWORDS

Convex Hull, 3D Envelope, Voronoi Diagram, Parallel algorithm

1 INTRODUCTION

Computing the convex hull of a set of hyper-points in $\mathbb{R}^d$ is a well-studied and recurring problem in Computational Geometry. T.M. Chan outlines an optimal, output-sensitive algorithm in [2] running in $O(n \log h)$ where n is the size of the input and h is the size of the output for $d = 2$ and $d = 3$. The recent large-scale adoption of multi-core architectures has unlocked n-fold speed-ups in the context of embarrassingly parallel problems. Still, bringing such run-time improvements in convex hull algorithms requires a partial-to-complete rethinking of how the problem is tackled. Based on [1], Staals provides in [4] a theoretical outline of various parallel algorithms used for indirectly computing convex hulls. Given a set of points $\mathcal{P} \subset \mathbb{R}^d$, the problem of computing the convex hull of $\mathcal{P}$ is reduced to stitching together the lower and upper 3-dimensional envelopes in the dual space $\mathcal{P}^\star$ defined by 2. Given $H = \mathcal{P}^\star$ and $n = |H|$, Staals shows in [4] that the lower and upper 3-dimensional envelopes of the set of planes H can be computed by means of the parallel main algorithm in $O(\frac{n}{p} \log n)$ time with p processors. Such algorithm is supported by two parallel sub-routines, namely the brute-force and batch-point-location algorithms that are shown to run both in $O(\log p)$ time with p^4 and $\max\{n, p^8\}$ processors respectively. Based on these assumptions, the goal of this project is to investigate the practical feasibility and, if possible, implement and benchmark the algorithms provided by Staals in [4] to establish how well the asymptotic complexities relate to the actual algorithms' runtimes as the number of processors p is increased. We further divide the main objective into the following tasks: first, investigate the practical feasibility of such algorithms by studying [4]; second, provide a sequential implementation of the brute-force and batch-point-location algorithms in a programming environment of choice; third, provide a visualization tool for visualizing the produced lower envelope; fourth, provide a parallelized version of the previously mentioned algorithms; fifth, provide a sequential and parallel implementation for the main 3d lower envelope algorithm; lastly, investigate, employing benchmarking, how the algorithms' runtimes relate to the claimed theoretical upper bounds in [4] as the number of processors p increases.

2 DEFINITIONS AND OVERVIEW

In this section, I introduce the theoretical fundations required to fully understand the context of the problem and the solution proposed by Staals in [4]. Next, a brief overview about the project's structure, the algorithms involved and how they relate to each other is provided.

2.1 Fundamental Definitions

2.1.1 Convex Hull. The convex hull $\text{Conv}(\mathcal{P})$ of a finite set of d-dimensional points $\mathcal{P} \subset \mathbb{R}^d$ is define as the smallest convex set that contains $\mathcal{P}$. A convex hull can be partitioned into upper and lower hulls: the upper hull contains all the vertices $v \in \text{Conv}(\mathcal{P})$ for which an upward, vertical ray defined at v doesn't intersect the hull; the lower hull contains all the vertices $v \in \text{Conv}(\mathcal{P})$ for which a downward, vertical ray defined at v doesn't intersect the hull.

2.1.2 Lower and Upper Envelopes. Let H be a set of n non-vertical planes in $\mathbb{R}^3$. The lower envelope of H is defined as

$$L_0(H) = \{(x, y, z) \mid (x, y) \in \mathbb{R}^2 \wedge z = \min_{h \in H} h(x, y)\} \tag{1}$$

The upper envelope $U_0(H)$ is define similarly, with the only difference that z is maximised among the available planes in H. $L_0(H)$ and $U_0(H)$ define a piecewise linear terrain in $\mathbb{R}^3$ and their 2-dimensional projection define a convex subdivision of $\mathbb{R}^2$. An example of such subdivision is shown in Figure 1 with H set to a random list of 25 non-vertical planes.

2.1.3 Duality Transformation. A duality transform is a bijective transformation mapping d-dimensional points into d-dimensional hyperplanes and vice-versa. The domain and image of such transformation are called primal (items identified with lower-case letters, such as r) and dual spaces (items identified with lower-case letters and a star, such as $r^\star$). The duality transformation used in this

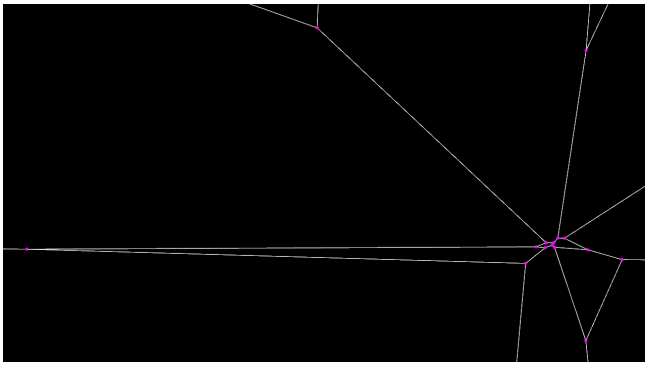

Figure 1: Convex subdivision of $\mathbb{R}^2$ induced by $L_0(H)$ with random H visualized with our tool (more information in Section 6)

project

$$\begin{aligned} p = (x, y, z) \Rightarrow p^\star \equiv c = ax + by - z \\ h \equiv ax + by + cz + d = 0 \Rightarrow h^\star = (a, b, -d) \end{aligned} \tag{2}$$

is defined in $\mathbb{R}^3$ and guarantees the reverse order property for planes with equation $ax + by - z + d = 0$. Let p and h denote a point and a plane in the primal space. The reverse order property guarantees that $p^\star$ is above $h^\star \iff p$ is below h. By "above" (+) and "below" (-), I refer to the sign of $p_x a + p_y b + p_z c + d$ where $(a, b, c, d) = h$ are the plane equation coefficients.

2.1.4 Envelopes-Hulls Relation. Given a set of points $\mathcal{P} \in \mathbb{R}^3$, the upper and lower hull of $\text{Conv}(\mathcal{P})$ are equivalent to $L_0(\mathcal{P}^\star)$ and $U_0(\mathcal{P}^\star)$ in the dual space. Consequently, an efficient envelope algorithm would also provide an indirect, yet efficient method for computing convex hulls.

2.2 Overview

In [4], Staals outlines a parallel algorithm for efficiently computing lower envelopes in $O(\frac{n}{p} \log n)$ time with p processors. Such an algorithm (the main algorithm from now on) employs randomization and divide and conquer techniques as explained in [4]. At each recursive step, the remaining problem to solve is computed with a batch-point-location algorithm and a random sampling scheme is used to sub-divide the problem into smaller chunks; once the size of the problem is sufficiently small, the brute-force algorithm is used to compute a solution. In the following sections, a data representation for envelopes and batch-point-location results is first proposed. Then, each algorithm in [4] is discussed while considering edge cases and implementation details. In the last section, various benchmarks are carried out to establish how the actual algorithms runtimes relate to their theoretical asymptotic complexities as the number of processors p increases.

3 DATA STRUCTURES

In this section, the data structures used to represent the output of the brute-force, batch-point-location and main algorithms are introduced. In particular, lower/upper envelope and batch-point result representations are discussed.

3.1 Envelope Representation

Adopting a proper envelope representation is a fundamental step towards implementing lower envelope algorithms as it directly affects how such algorithms will be designed. Also, it is essential to consider which operations will be applied on the data structure and what computational and memory complexities are expected. There are four types of possible envelopes: $L_0(H) = \emptyset$ if $H = \emptyset$; $L_0(H) = \{p \in H\}$ if all the planes in H are parallel to each other; $L_0(H) = \{\text{Line with both ends at infinity}\}$ if no triple of planes produce a vertex (and it is not the previous case). In Figure 2, all the possible types of envelopes, except $\emptyset$, are listed. An empty envelope is simply empty, an envelope defined by just one plane is simply the id of that plane and an envelope with edges is defined as a sorted sequence of directed edges. Based on these assumptions, an envelope is represented as a union of different types: empty, integer and a list of edges. Note that the endpoints of each edge change the semantics of that edge: an edge of two infinite points is a line, while an edge of one infinite and one finite point is a ray and so on. A comprehensive list of such cases is shown in Figure 3. Also, each subset of edges defines a bounded or unbounded face on the envelope: a face is said to be bounded if it is enclosed within a convex polygon defined by a subset of edges in the envelope, otherwise unbounded. In order to uniquely identify each face in the envelope, every edge is required to store additional information stating which plane defines the envelope to its left.

1 **Union** *Envelope* **contains either**
  Empty: $\emptyset$
  Integer: Plane Id
  $\prec$ **Sorted Sequence:** Edges
2 **end**

Algorithm 1: Envelope represented as a union of types

1 **Struct** *Edge* **contains**
  Point: Start
  Point: End
  EdgeType: Type
  Integer: Lowest Left Plane Id
2 **end**

Algorithm 2: Envelope edge representation

Our data structure should allow two operations: fast neighbour retrieval and face triangulation. Because of these requirements, the sequence of edges $L_0(H)$ is sorted according to a total order $\prec$ defined as

$$\begin{aligned} x = (1, 0, 0) \\ \overline{\text{ab}} \prec \overline{\text{cd}} \Leftrightarrow a < c \vee (a = c \wedge \measuredangle(a + x)ab < \measuredangle(c + x)cd) \end{aligned} \tag{3}$$

on every pair of edges. The result is two-fold: first, $\forall v \in H$, the sequence of incident edges from v are stored contiguously so that the entire list of neighbours can be retrieved fast; second, the sequence is locally sorted around each v in counter-clockwise order so that an easy triangulation routine can be used (more on that in Algorithm 11).

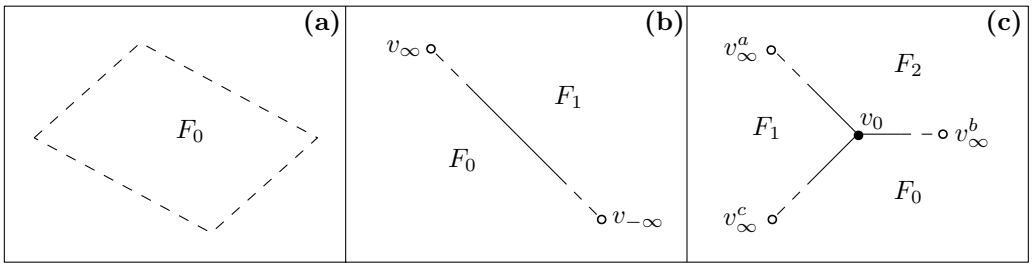

Figure 2: (a) one-plane envelope, (b) one-line envelope projected on the xy-plane, (c) finite-vertex envelope projected on the xy-plane

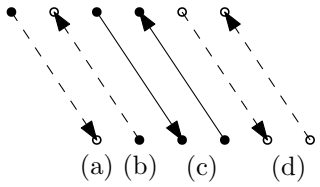

Figure 3: (a) Half-edge with finite start vertex, (b) Half-edge with finite end vertex, (c) Segments, (d) Lines

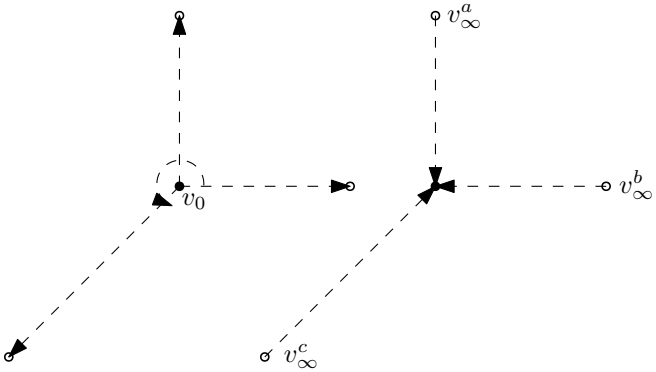

Figure 4: Envelope representation for case (c) in Figure 2

3.2 Point Location Representation

Given a set of planes H and a set of query points Q, the batch-point-location algorithm computes the z-sorted sequence of planes Z from H above $\forall q \in Q$. It is important to note that H has different z-sorted sequences based on the location of q (more on that 4.2) but also, different $q', q'' \in Q$ may have Z', Z'' sequences sharing the same z-order. In order to achieve the main algorithm's runtime complexity, the sorted sequence of planes above each query point must be retrieved in $O(1)$. I provide the batch location representation in Algorithm 3. For each point q, a pair of integers i, j is used to locate the sequence of z-sorted planes and the first plane above q in that sequence. Given that multiple query points may share the same sorted sequence, a common list is used to store all the z-sorted sequences referenced by at least a query point in Q.

1 **Struct** *Query Result* **contains**
  Integer: Sorted Sequence Id
  Integer: First Plane Id Above
2 **end**
3 **Struct** *Points Location* **contains**
  List: Z-Sorted Planes Sequences
  List: Query Results
4 **end**

Algorithm 3: Point Location result representation

4 ALGORITHMS

In this section, the brute-force, batch-point-location, and main algorithms are outlined and discussed. In particular, a theoretical and practical point of view is provided for each algorithm.

4.1 Brute Force Algorithm

4.1.1 Theoretical Fundations. Given a set of non-vertical planes H, the brute-force algorithms exhaustively search for $L_0(H)$. The algorithm proposed by Staals is found in [4]. A worst-case sequential execution of this algorithm requires $O(n^3)$ work, as finding the valid vertices of the envelope is the most expensive operation. At this point, there is one important point to cover: what is the algorithm supposed to do with unbounded edges? Computing unbounded edges is a fundamental issue because the algorithm does not consider vertices at infinity. It implies that only the subset of bounded faces in $L_0(H)$ can be computed. Because of this fundamental limitation, I designed an alternative algorithm (shown in Algorithm 5) which also computes vertices at infinity and thus unbounded edges. A worst-case sequential execution of this algorithm requires $O(n^4)$ work and $O(n)$ more expensive than the first proposed method. The additional work is introduced by the required exhaustive search to compute the sets

$$\begin{aligned} S &= \{l_i^k \mid \exists p_i, p_k \in H \mid p_i \cap p_k = l_i^k\} \\ V_f &= \{v_i^k \mid \exists l_i, l_k \in S \mid l_i \cap l_k = \{v_i^k\}\} \\ V_- &= \{v_i \mid \exists\, l_i \in S \mid \forall\, v_k \in l_i \; v_i < v_k\} \\ V_+ &= \{v_i \mid \exists\, l_i \in S \mid \forall\, v_k \in l_i \; v_i > v_k\} \end{aligned} \tag{4}$$

where V_-, V_+ are the left and right sets of vertices at infinity along all lines in S. The sequence $V = V_f \cup V_- \cup V_+$ is sorted in lexicographical

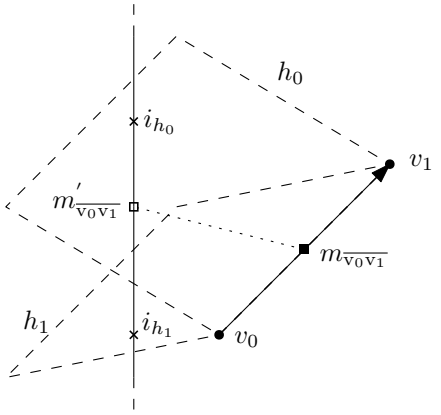

Figure 5: Locating the lowest plane for segment $\overline{v_0v_1}$

order and the predicate

$$L(u,v) = \frac{u_z + v_z}{2} \leq \min_{h \in H} h\left(\left[\frac{u+v}{2}\right]_{xy}\right) \tag{5}$$

is tested for all consecutive pairs of vertices $(u, v) \in V$, producing the set of edges

$$E = \{(u, v, H_{\text{left}}(u,v)) \mid u, v \in V \wedge u_i = v_i \wedge L(u,v)\}$$

$$H_{\text{left}}(u,v) = h \mid h \text{ generates } (u,v) \text{ while being its lowest left plane} \tag{6}$$

Given that there could be multiple consecutive edges $(u, v, h), (v, o, h)... \in E \mid u_i = v_i = o_i = ...$, a reduction operation on E is performed until no more consecutive edges along the same line are found. Finally, $L_0(H)$ is the sorted sequence E according to $\prec$. An algorithm for $H_{\text{left}}(u,v)$ and its visual representation are shown in Algorithm 4 and figure 5. The edge cases mentioned in **??** also apply to this algorithm.

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

Data: $e = \overline{v_0v_1} \in E$ and set $H' \subset H$ generating e
Result: $H_{\text{left}}(v_0, v_1)$

```
Z ← (0, 0, 1);
T ← 1000;
m_{v0v1} ← (v0 + v1)/2;
m'_{v0v1} ← −(ê × Z) · T;
l ← Vertical line passing through m'_{v0v1};
return arg min_{h∈H'} (l ∩ h)_z
```

4.1.2 Implementation Details. Even if the algorithm can be considered embarassingly-parallel, there are some important aspects that should be taken into account. Lock-free, thread-safe operations on arrays require two ingredients: first, no two threads should write or read and write to the same location; second, the array memory shouldn't get resized while threads are working on it. Atomic counters and local memory to each core is used to guarantee the first point: in particular, a relaxed memory order is used when updating an atomic variable, so that inter-core synchronization is not required. The second point is guaranteed to be true by allocating a memory chunk sufficiently large to store the worst-case output size.

Algorithm 5: Brute-Force Lower Envelope Algorithm

Data: A set of non-vertical planes H
Result: $L_0(H)$

1. Compute the set of unique lines S generated by the set of planes H;
2. Compute the sequence of lexicographically sorted vertices V from S while considering vertices at infinity (as shown in Figure 6);
3. Filter out all the edges (i.e., all the consecutive pairs of vertices in V along the same line as shown in Figure 7) for which at least one plane in H below it exists;
4. Merge consecutive edges along the same line and collect its lowest left plane (as shown in Figure 8);
5. Sort the final sequence of edges according to $\prec$;

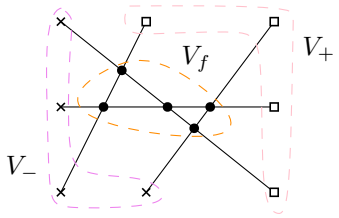

Figure 6: Sets of vertices

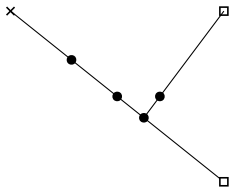

Figure 7: Segments satisfying predicate $L(u,v)$

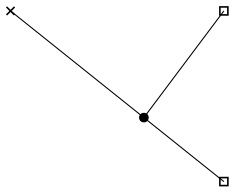

Figure 8: Final envelope edges

4.2 Batch Point Location Algorithm

Given a set of non-vertical planes H and a set of query points Q, the goal is to find the z-sorted sequence of planes $(h_i, .., h_n) \mid q_z \leq h_i(q_{xy}) \leq .. \leq h_n(q_{xy})$ for each query point $q \in Q$. The algorithm based on Staals' proposal is shown in Algorithm 6. The algorithm implemented in this report is slightly different from the

one proposed by Staals for two main reasons: first, the number of sorted sequences of planes is lazily computed; second, for each query point, the computation is split into three different phases, so that the parallelization is more straightforward to achieve.

Lemma 4.1. *Let H be a set of non-vertical planes and Q a set of query points. The batch-point-location algorithm sequentially computes the sorted sequences of planes in H above each query point in Q in*

$$O(\max\{|Q|, |H|^4\} \log(|H|^4) + |H||Q| \log(|H|)) \tag{7}$$

work for the worst-case scenario.

Proof. The steps provided in Algorithm 6 are performed with the following amount of steps: Step 1 is performed in $O(|H|^2)$; step 2 is performed in $O(|H|^4 \log |H|^4)$ with a quicksort algorithm; step 3 is performed in $O(|Q| \log |H|^4)$ with a binary search algorithm; step 4 is performed in $O(|Q| \log |S| + |Q||H| \log |H|)$ with a binary search algorithm applied to the list of lines S with $O(|H|^2)$ items generated from H; step 5 is performed in $O(|Q|)$ with an hash map providing constant access time. ∎

As discussed in [4], the goal with the batch-point-location algorithm is to have an algorithm fast when H is small and Q is big. Such property is useful for the main algorithm (discussed in Section 4.3) as it works with sampled instances of the problem producing a limited amount of planes and many query points. The total order $\prec_i$ on S for slab i starting at $v_x^i \in V$ is defined as

$$\nabla l \in S = \text{Slope of line } l$$

$$\prec_i (l, m) = \begin{cases} l & l(v_x^i) < m(v_x^i) \\ l & l(v_x^i) = m(v_x^i) \wedge \nabla l < \nabla m \\ u & \text{otherwise} \end{cases} \tag{8}$$

and produces the sorted sequence of lines S_i. Two important corner cases are now discussed: what happens when $S = \emptyset$ or $|S| > 0 \wedge V = \emptyset$? Since the set S is empty, then all planes in H are parallel to each other. Consequently, $\mathbb{R}^2$ can be interpreted as an infinite slab with just one unbounded face, thus the algorithm can jump straight to step 5. Conversely, if V is empty but $|S| > 0$ then all the lines in S are parallel to each other. Consequently, $\mathbb{R}^2$ is partitioned into $|S|+1$ slabs with one unbounded face each. The problem is thus slightly simplified: given a point q, once the i-slab containing q is located in step 3, the algorithm can jump straight to step 5. Finally, two additional edge cases that may arise while searching for the face of a given query point must be addressed: given a query point $q \in Q$, what happens if (a) $q_x = v_x^i$ or (b) $\exists l \in S \mid l \cap q_{xy} \neq \emptyset$? In case (a), two possibilities arise: if $q \in V$, then H_i^q contains at least three entries with the same z-value and so the sorted subsequence must contain all such planes; otherwise, if $\exists l \in S \mid l$ is vertical $\wedge\ l \cap v_x^i \neq \emptyset$ then q is a point along the vertical line passing through v_x^i; in this case, the $k-$th segment on l has a different H_i^k. This last case is the same as (b), as each line passing through the i-slab can be considered as a segment, thus each having a different H_i^l.

4.2.1 Implementation Details. In practice, computing a full slab is very expensive: $O(|S| \log |S|)$ work is required for the sorted sequence of lines to be computed, while $O((2(|S|+1)+k) \cdot |H| \log |H|)$ work is required for the z-sorted sequences of planes to be sorted at each possible location within the slab (i.e. a face, a vertex or a segment). Because of this reason, various lazy techniques are employed: first, the sorted sequence S_i is computed only if there is at least one point in Q within i-slab; second, $H_i^{(f|l|v)}$ is computed only for faces, segments and vertices for which at least one point in Q is located there; third, a caching system collects every $H_i^{(f|l|v)}$ computed so that the same work is not repeated. A sequential binary search routine eliminates the additional overhead introduced by recursion. The lock-free, parallel implementation was made possible thanks to the separation of computation in three different steps: first, the slab is located for every query point; second, each required slab is computed; third, all the z-sorted sequences of planes required are stored in the cache. Finally, the result for each query point is written concurrently.

Algorithm 6: Batch Point Location Algorithm

Data: A set of non-vertical planes H and a set of points Q.
Result: The sorted lists of planes above each point in Q.

1. Compute the set of unique lines S generated by the set of planes H and project them onto $\mathbb{R}^2$;
2. Compute the lexicographically sorted sequence of vertices V from S. Each pair of consecutive vertices $v, u \in V$ defines a slab, an interval region $[v_x, u_x)$ in which no intersections occur. Additionally, two infinite intervals are defined for the vertices $v = \min_{v \in V} v_x,\ u = \max_{v \in V} v_x$, $(-\infty, v]$, $[u, \infty)$ (as shown in Figure 9;
3. For each query point in Q, locate the slab containing it by means of binary search. Then, for each i-slab containing at least one query point, compute the sorted sequence of lines S_i according to $\prec_i$. S_i partitions the i-slab in $|S_i| + 1$ faces: in particular, the bottom-most and upper-most faces are unbounded while the other $|S_i| - 1$ faces are bounded;
4. For each query point q in Q, use binary search to locate face f in the i-slab containing q. If q is the first point to be found in face f, then shoot a vertical ray and compute the z-sorted sequence of planes H_i^f in increasing order (as shown in Figure 10); otherwise, fetch the sequence of sorted planes from a previous point in f;
5. Allocate an output array A of size proportional to $|Q|$. For each query point q in Q, locate the first plane in H_i^f above q and store the sorted subsequence in A;

4.3 Main Algorithm

Given a set of non-vertical planes H, the main algorithm computes $L_0(H)$ by jointly using algorithms 4.1 and 4.2. A worst-case sequential execution of the algorithm proposed by Staals in [4] would require $O(|H| \log |H|)$ work. Two main limitations are required to be mentioned: first, the algorithm complexity is based on the double sampling scheme used in the algorithm; unfortunately, for the p-sample $S \subset H$ to have a sufficient amount of items, the input set H is required to be large $|H| > 10^5$; based on the algorithms presented in this report, such size is practically unfeasible to achieve because the time and space used would be dominated by the inefficiencies of the brute force algorithm; second, the unbounded faces of the

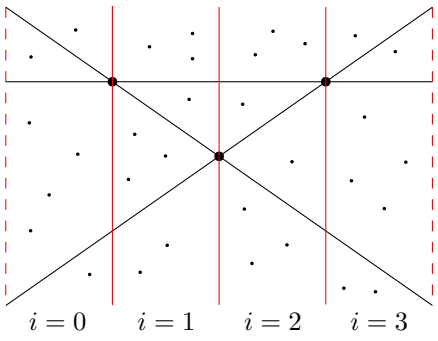

Figure 9: $\mathbb{R}^2$ slab partition generated by three intersecting lines

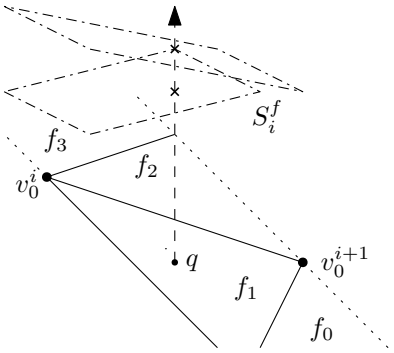

Figure 10: Z-Sort for a point q in slab $i = 1$

top-most envelope may not be large enough to contain all planes intersecting its prism (more information about the issue are found in the implementation details section of this algorithm). Because of these reasons, Algorithm 10 is proposed. This algorithm uses a single scheme sampling on the input set H and adjusts with Algorithm 7 the unbounded faces to mitigate the issue previously reported. The main drawback of this algorithm is that there is no upper bound function nor constant recursion depth proof. If $L_0(S)$ is classified as either single-plane or single-line (as shown in Figure 2), then $C_{L_0(S)}$ contains all the planes in $H - S$ intersecting the half-space induced by $L_0(S)$; a solution is provided in Algorithm 8. From $H - S$, the subset of conflict planes for each prism in $L_0(S)$ is computed. Assembling $L_0(H)$ requires an adjacency-aware and embarrassingly-parallel procedure that is shown in Algorithm 9: given a pair of consecutive entries in E_{i-1}, adjacency is guaranteed; $O(\frac{|E_{i-1}|}{2})$ merging operations $\text{Merge}_\triangle$ defined as

$$\text{Merge}_\triangle(u, v) = \begin{cases} \emptyset & L_0(C_u) = \emptyset \wedge L_0(C_v) = \emptyset \\ L_0(\{p_i\} \cup C_v) & L_0(C_u) = i \\ L_0(\{p_j\} \cup C_u) & L_0(C_v) = j \\ \text{Merge edges as in [4]} & \text{otherwise} \end{cases} \tag{9}$$

are iteratively performed in parallel until one single prism is left.

4.3.1 Implementation Details. The provided implementation should not be considered reliable as the time was insufficient to test and debug the code. Still, it could be used as a starting point for a reliable and fast implementation. Some parts of the code are parallel while others are left sequential because, again, the time was insufficient. In practice, a point at infinity is still finite but is placed very far along the same half-edge. This introduces a subtle edge case that may arise when the top-most $L_0(S)$ is computed: there could be vertices not sufficiently far to include all the intersecting planes from $H - S$ for a specific prism. This problem is shown in Figure 11, where a plane intersects the prism after the vertex at infinity v_∞^b. An algorithm tackling this limitation in $O(n \log n)$ where $n = |L_0(S)|$ is proposed in Algorithm 7. Essentially, each prism missing one or more planes from its conflict list is moved to the fartest intersection with such planes along the half-edge, so that they are all included. This adjustment is only required for the top-most envelope because all the sub-envelopes will be correctly bound by their prisms, so no plane can miss the conflict set. The input set is restricted to planes having coefficient $c = -1$.

Algorithm 7: Adjust vertices at infinity

Data: $K = H - S, L_0(S)$
Result: Adjusted $L_0(S)$
Partition and sort half-edges in $L_0(S)$;
foreach *Disjoint pair of half-edges* $e_{in}, e_{out} \in L_0(S)$ **do**
  $P \leftarrow$ Collect all $h \cap e_{\text{out}}\ \forall h \in K$;
  $e^1_{\text{out}}, e^0_{\text{in}} \leftarrow \arg\max_{p \in P} \|p - e^0_{\text{out}}\|$;
end
return $\prec L_0(S)$;

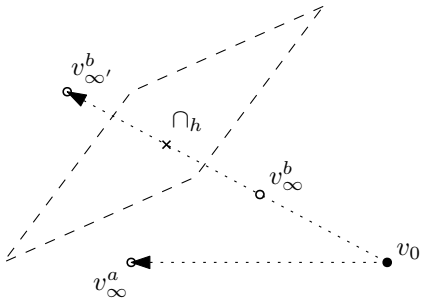

Figure 11: Adjust infinity points

Algorithm 8: Conflict Lists edge cases

Data: Planes in $L_0(S) = K$ and $P = H - S$
Result: Planes intersecting halfspace generated by $L_0(H)$
$S \leftarrow \emptyset$;
foreach *Plane* $p \in P$ **do**
  if $\exists h \in K \mid h_d \geq p_d \wedge \hat{h} = \hat{p}$ **then**
    $S \leftarrow S \cup \{p\}$;
  end
end
return S;

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
    $T[a], T[b] \leftarrow \text{true}, \text{true}$;
    $m \leftarrow$ 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 10: Main algorithm

Data: A set of non-vertical planes H.
Result: $L_0(H)$
If $|H| <$ Threshold then compute and return $L_0(H)$ by using the brute force algorithm;
Otherwise, sample $S \subseteq H$ with probability $\frac{1}{|H|^{\frac{1}{8}}}$;
Compute $L_0(S)$ and triangulate it with Algorithm 11. Then, compute the set of unique prisms $P_{L_0(S)}$ with Algorithm 12;
For each prism $p_i \in P_{L_0(S)}$, compute the conflict set of planes $C_i \subseteq H - S$ using the batch point location algorithm in dual space (as explained in [4] and 2) and go to step (1) by setting $H = C_i$;
At the end of each recursive step, assemble $L_0(H)$ by iteratively merging pairs of adjacent envelopes (as shown in Algorithm 9) until no more pairs are found. Return $L_0(H)$;

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 $\prec E$;

Algorithm 12: Compute Unique Prisms in $L_0(H)$

Data: A triangulated $L_0(H)$
Result: The list of unique prisms
$T \leftarrow$ An empty array;
foreach *pair of consecutive edges* $e, p \in L_0(H)$ **do**
  $v_0, v_1, v_2 \leftarrow e_{\text{start}}, e_{\text{end}}, \text{null}$;
  if $e_{start} = p_{start}$ **then**
    $v_2 \leftarrow p_{\text{end}}$;
  end
  else if $e_{start} \notin V_f$ **then**
    $s \leftarrow$ First edge from past sequence;
    $v_2 \leftarrow s_{\text{end}}$;
  end
  else
    continue;
  end
  Sort the vertices v_0, v_1, v_2;
  $T \leftarrow$ Insert triangle $\langle v_0, v_1, v_2 \rangle$;
end

5 DEVELOPMENT ENVIRONMENT

In this section, I describe the project structure, the tools and the development practices adopted. Also, the external libraries supporting the codebase are listed and briefly discussed.

5.1 Project Structure

The project is based on CMake 3.1+ build-system and Vcpkg for dependency management. The building process is currently defined and tested for Microsoft Visual Studio 2022 and the MSVC++ compiler using C++ 17. The project is based on the following GitHub repository [3].

5.2 Third-Party Libraries

5.2.1 CGAL. CGAL is a software project providing access to efficient and reliable geometric algorithms through a C++ API.

5.2.2 HPX. HPX is a library for concurrent and parallel distributed computing. It provides an STL-like interface and different execution policies: sequential, parallel and unsequenced parallel (joint usage of vectorization and parallelization techniques).

5.2.3 Intel One TBB. OneTBB is a flexible performance library that simplifies the work of adding parallelism to complex applications.

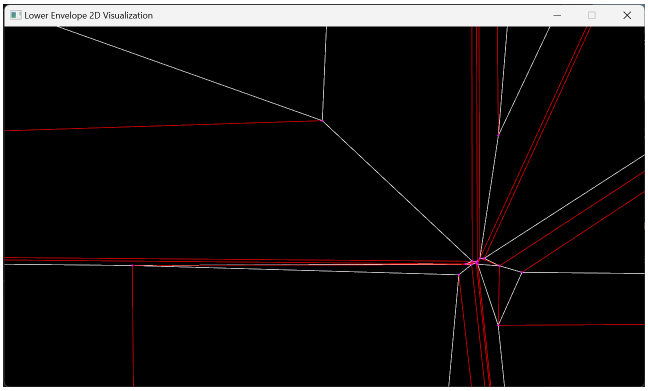

Figure 12: Triangulated Envelope Visualization (zoom in)

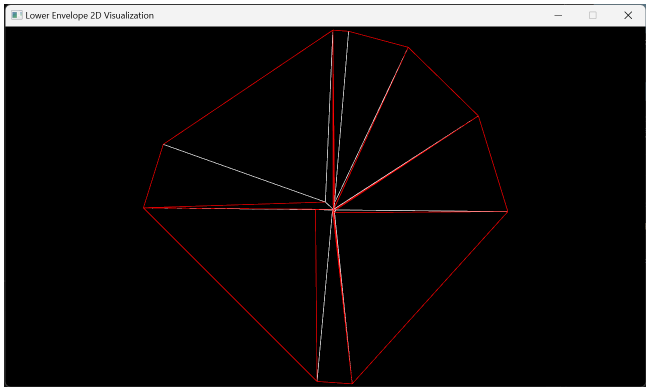

Figure 13: Triangulated Envelope Visualization (zoom out)

It provides low-level APIs and complete control over the implementation to the programmer. Because of some memory-related limitations discussed in the benchmark section, I decided to use oneTBB and reimplement all the parallel algorithms to get better performances (more on that in the benchmark section).

5.2.4 Catch2. Catch2 is a C++ framework providing unit testing and benchmarking capabilities.

5.2.5 SFML. SFML is a portable graphics and media API written in C++, which can be considered an object-oriented SDL library.

5.3 Development Life-cycle

Given the nature of the project and the fact that bugs can be very difficult to spot, I decided to follow a mixture of TDD and CI/CD development lifecycle. In particular, for each algorithm implemented, I followed the following steps: first, I defined and implemented a series of test cases with Catch2 that the algorithm must be able to pass; each test case is thought to cover a specific edge case; then, the actual algorithm was implemented and tested until all tests were passed successfully and occasionally, some more tests were added. Once this process is done, the algorithm is left running for 100 different problem instances. If all tests are passed, then the current change list containing the new algorithm can be pushed to the master git repository. Otherwise, further work is required until such condition is met.

6 LOWER ENVELOPE VISUALIZATION

A tool for visualizing the convex subdivision of $\mathbb{R}^2$ (XY-plane) induced by $L_0(H)$ is provided within the codebase. Examples of such a visualization are shown in Figures 12 and 13: the envelope's edges are shown in white, while the additional edges added with the triangulation Algorithm 11 are shown in red. The user can zoom in, zoom out and move around the XY-plane with its mouse. This tool proved to be particularly helpful while debugging the code in the early stages of the project.

7 BENCHMARKS

Benchmarks are performed on a Windows laptop equipped with an I7-12700H with 14 parallel cores and 16GB of RAM. CPU frequency scaling is turned off using the software ThrottleStop because it could affect the tests. Two algorithms are tested: the brute-force and batch-point-location algorithms. The main algorithm is kept out of the benchmarking phase because it is still incomplete and needs careful testing before it is benchmarked. The first parallel implementations tested were based on the HPX library. Optimal parallelization for complex algorithms usually cannot be achieved using high-level APIs. On the other hand, designing optimized parallel algorithms is a time-consuming and challenging task. Because of this, we decided to trade off parallel efficiency with time. Unfortunately, the performances of both algorithms were heavily affected by I/O and memory latencies, as well as false sharing across processors. Such bottleneck showed a performance penalty for each additional core used during the computation. Because of the vast amount of slabs generated even with 60 input planes and the search procedure based on binary search, that is, highly cache inefficient, the batch-point-location algorithm suffered the most in terms of performance. The main common bottleneck for both algorithms was the intersection points computation from the list of lines generated by the input set of planes: the number of input planes n and the size of the intersection points set V are related by $|V| = O(n^4)$. For an input set containing $n = 150$ planes, around 62 million intersection points would be involved. Given that these points must be lexicographically sorted and unique to be used, having an efficient parallel sorting algorithm is critical. HPX provides a parallel sorting algorithm that performed very poorly in our case. The main bottleneck is again related to I/O and memory latencies introduced by the sorting algorithm. In particular, when using a thread pool with multiple CPU cores, the same thread should be assigned to tasks operating on continuous memory deterministically so that the CPU can perform efficient caching. I tried to tackle such a sorting problem with a custom, in-place, parallel merge-sort algorithm using the C++ implace_merge routine. Unfortunately, benchmarks for this code did not provide any substantial speed-up (less than 2%). Memory deallocation time was another critical problem in the batch-point-location algorithm. We measured that around 40% of the execution time was used for deallocating the slab data structure: the batch-point-location algorithm run with $|H| = 100$ required ≈ 18 seconds out of ≈ 45 seconds of execution time to deallocate the slab data structure used ($\approx$ 8GB). In this case, the reason for such a slowdown cannot be entirely motivated. Undoubtedly, the

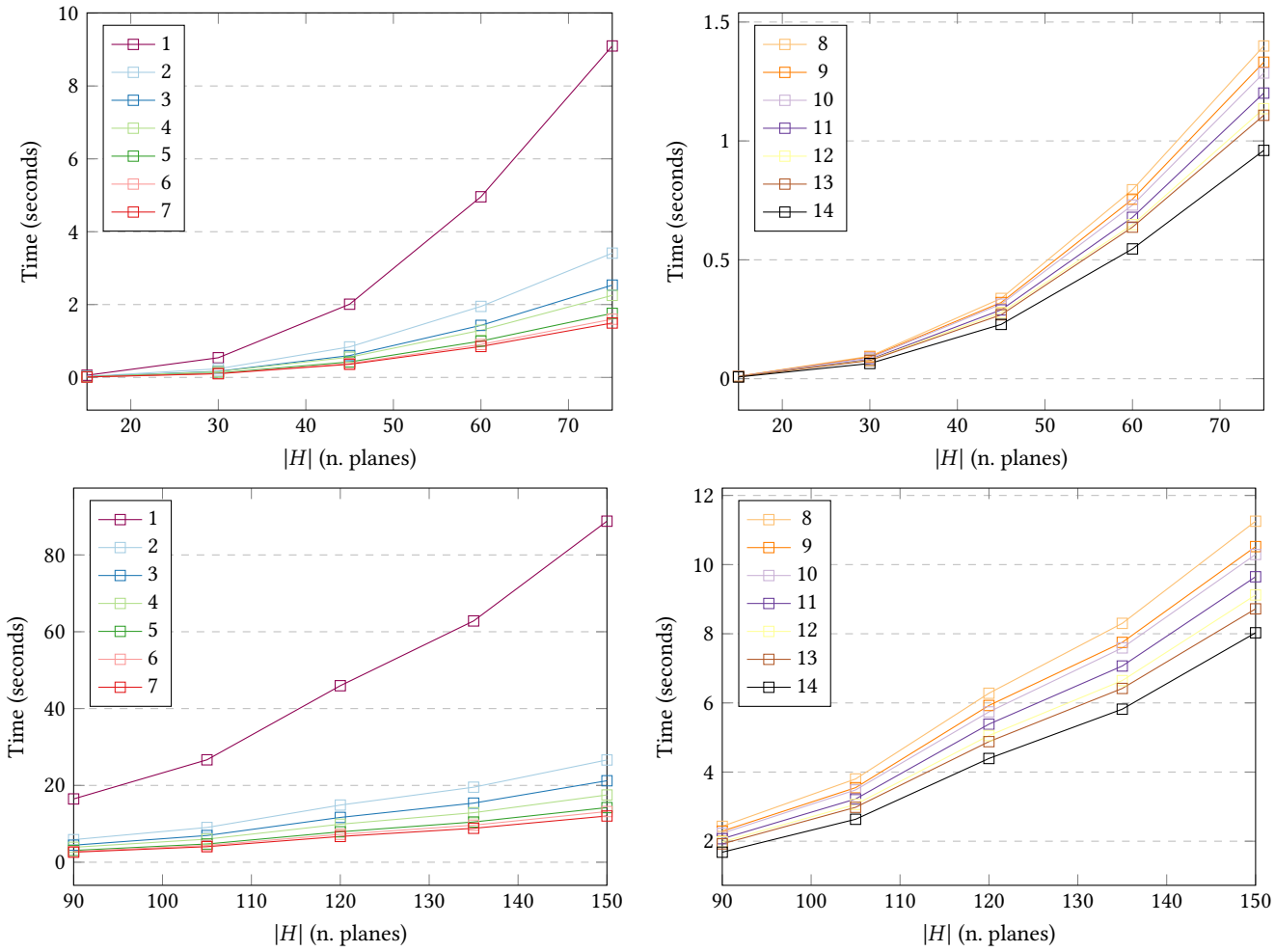

Figure 14: Brute Force algorithm benchmark with varying $|H|$

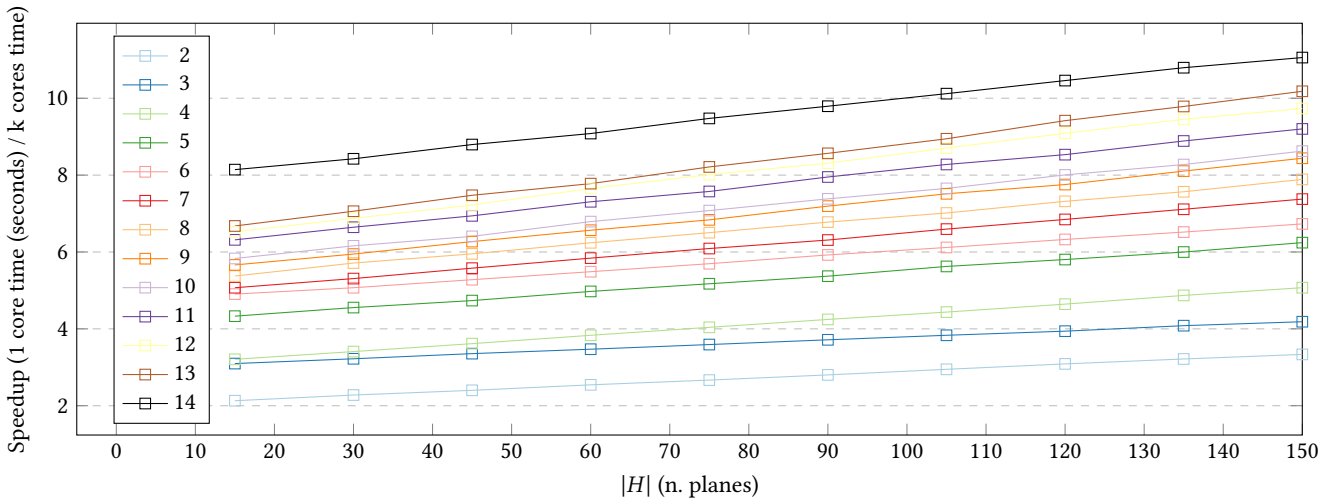

Figure 15: Brute-force algorithm parallel speed-ups

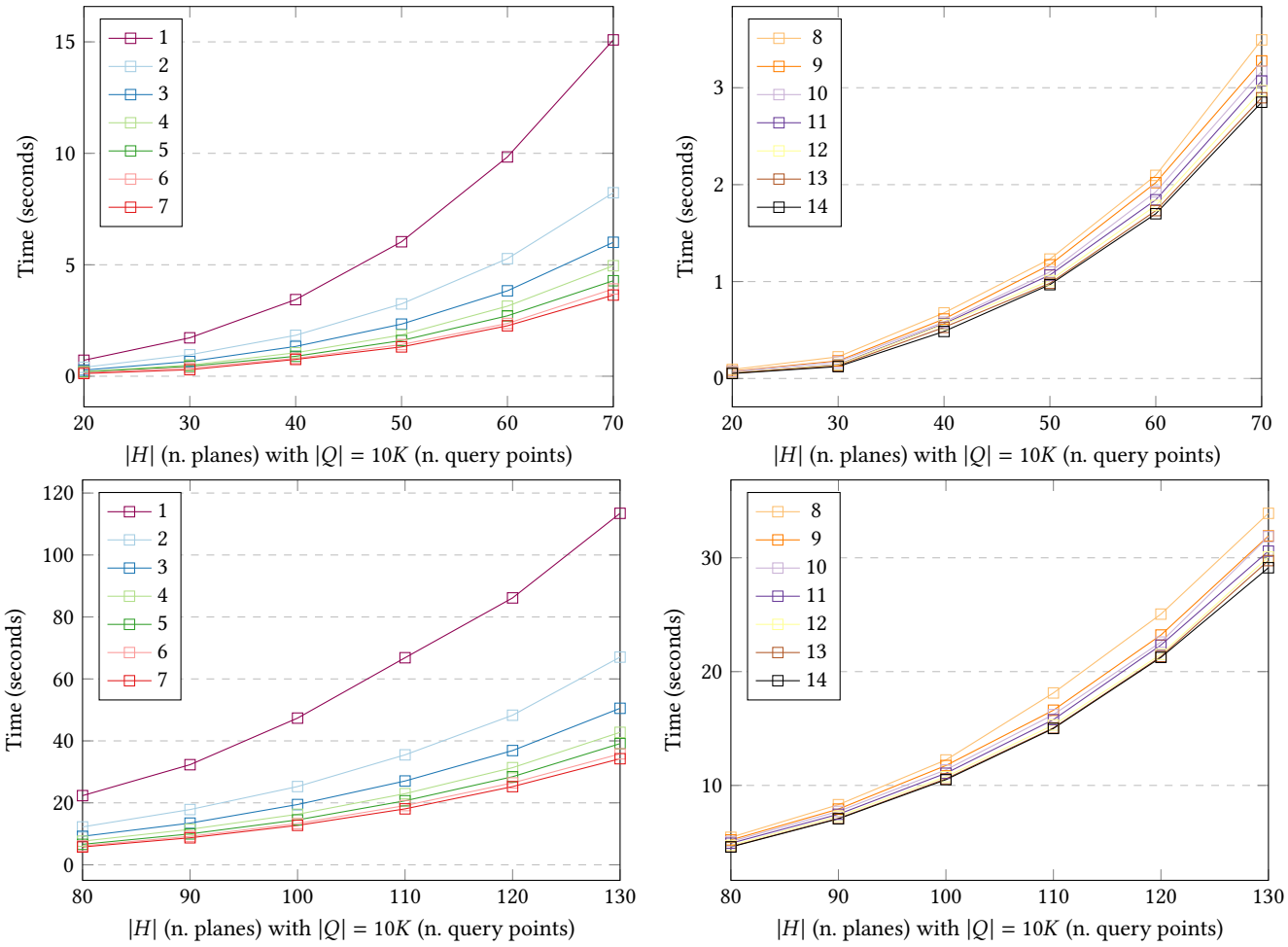

Figure 16: Batch Point Location algorithm benchmark with varying $|H|$

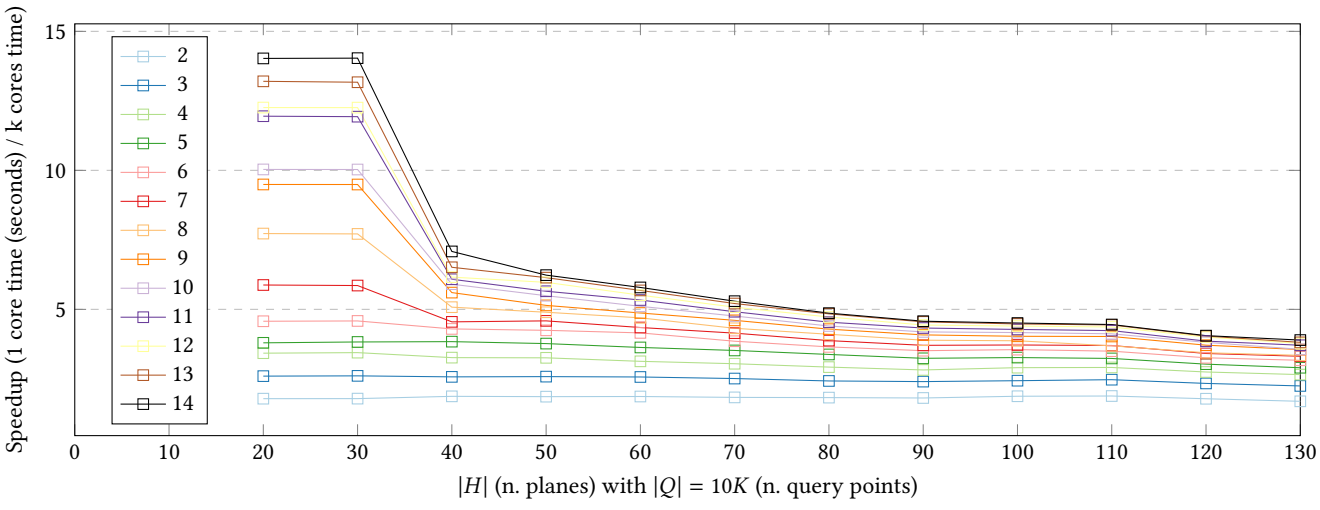

Figure 17: Batch-Point-Location algorithm parallel speed-ups

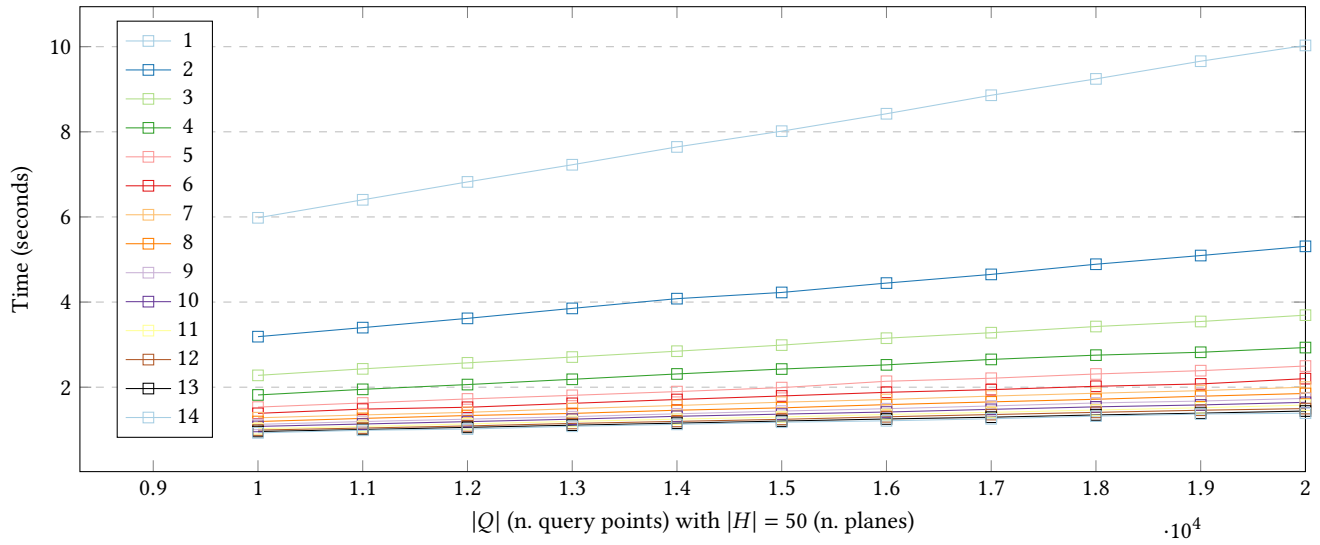

Figure 18: Batch-Point-Location algorithm benchmark with varying $|Q|$

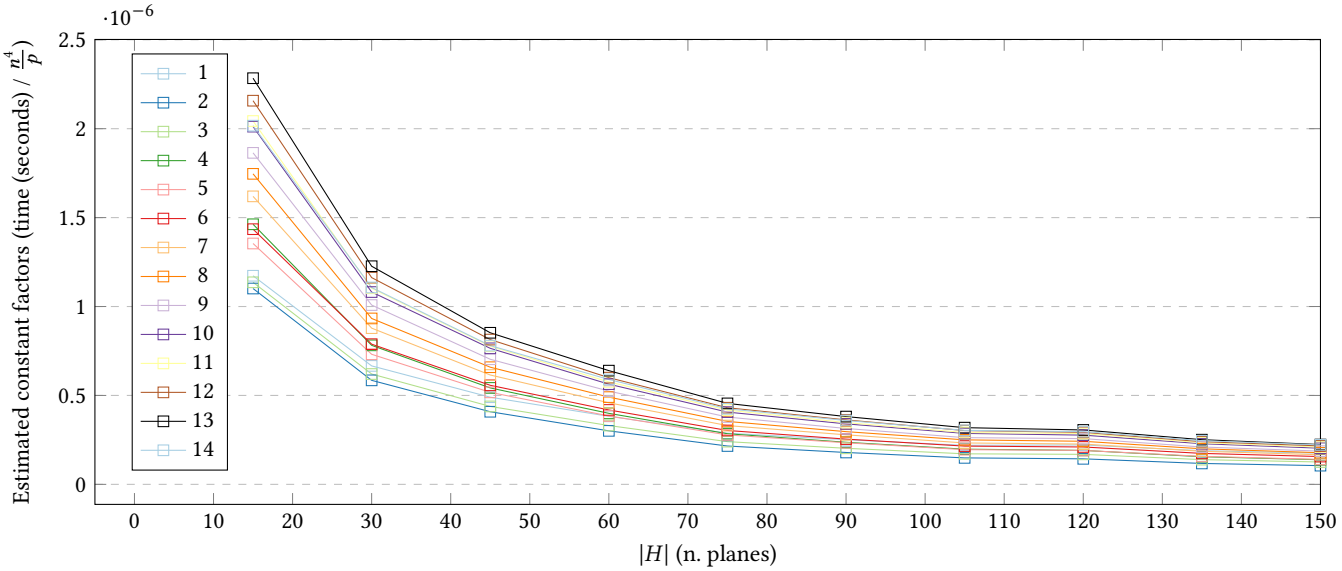

Figure 19: Brute-force algorithm theoretical factors

way CGAL's exact data types are laid out in memory does not help: Each item is represented as a handle to another memory location, thus introducing one level of indirection and limiting hardware from contiguously deallocating an entire block of memory. A profiling session revealed that, in fact, most of the deallocation time is spent within CGAL exact types' deallocation routines. CGAL's exact data types guarantee exact computations but use more computational resources, around 25% more. Therefore, using primitive data types such as float or double would significantly improve the performance of current algorithms. After researching these issues, I replaced HPX in favour of Intel OneTBB for code parallelization. There are multiple reasons for such a change: first, it gives the programmer complete control of the parallelization logic; second, many cache-related optimizations are more straightforward to achieve and faster in practice, as Intel designed the library to get most of the performances out of Intel CPUs; third, OneTBB is well-documented online thus reducing the development time required for the refactoring. The resulting implementations were then tested and benchmarked. The results are shown in Figure 14 and 16. In Figure 15, the brute-force algorithm run with 14 cores and 150 input planes shows around 11x performance improvement compared to the sequential version. Parallel performances improve as the number of cores and the input size increases. In Figure 17, the batch-point-location algorithm run with 14 cores and 130 planes

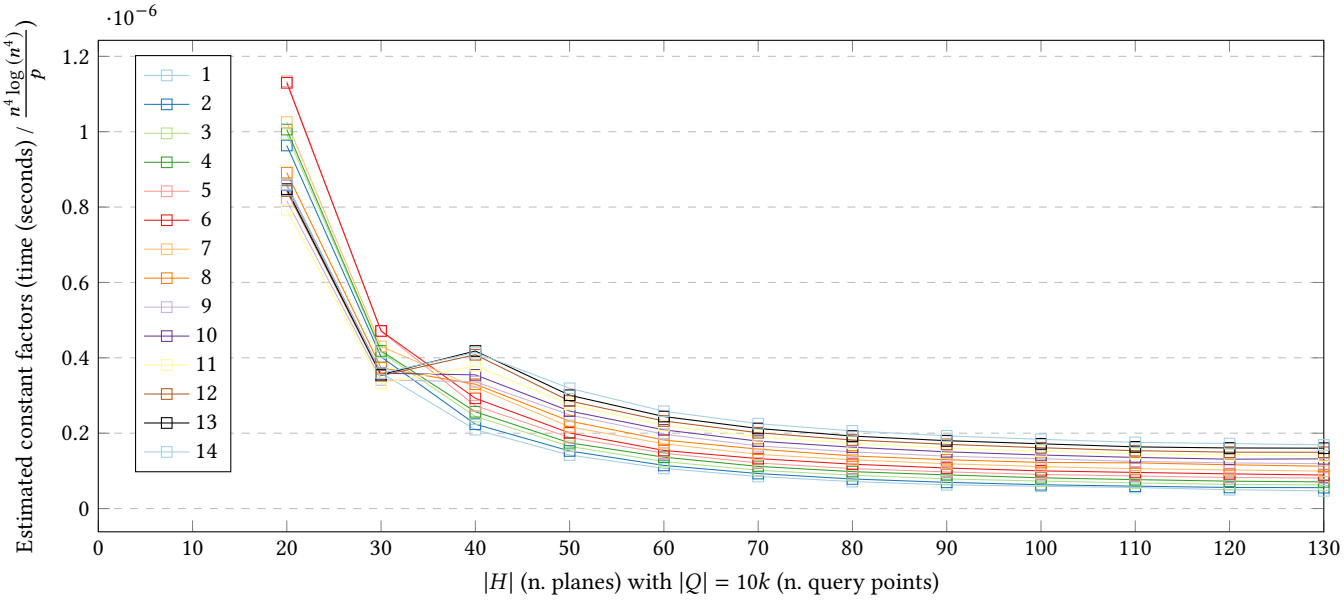

Figure 20: Batch-Point-Location algorithm theoretical factors

(with query points size fixed to 10k) shows around 5x improvement compared to the sequential version. A considerable performance drop is recorded for $n = 40$, while subsequent tests show minimal performance degradation with an overall efficiency decrease. A profiling session with Intel VTune showed that from $n = 30$ to $n = 40$, the amount of cache misses increased by around 70%. This is due to the non-linear relationship between the number of input planes and the number of intersection points that make the binary search lookup for each query point very expensive since all the data cannot fit into the CPU cache anymore. Nonetheless, the new OneTBB implementation successfully reduced I/O latencies thanks to various key optimizations.

7.0.1 Memory Allocation/Deallocation Overhead. OneTBB provides multiple types of memory allocators that can benefit a program's runtime. In particular, all the memory allocations and deallocations in the program are now handled by OneTBB's cache-aligned and scalable allocator. Introducing this change was straightforward: thanks to the tbbmalloc_proxy library, all the dynamic memory functions backends are replaced with OneTBB's allocator. This minor change reduced the runtime by approximately 30%.

7.0.2 Threading-Pool and Caching Overhead. Issuing the right amount of tasks to the threading pool requires a careful subdivision of the problem. Both algorithms use nested loops issuing one task per item pair; the efficiency of such division may be sufficient for very long operations but highly inefficient for quick ones. Because of this reason, we switched to a tile-based subdivision of the work. In particular, given n planes or lines, the amount of work required to compute the sequence of intersections is reduced from n^2 to $n + \sum_{i=0}^{n}(n-i)$ and the remaining work divided in squared tiles of side $\frac{n}{\#\text{cores}/2}$. OneTBB provides automatic and cache-compliant chunking through the affinity_partitioner object. Tiling for loops improved both algorithms' runtime by around 20%.

7.0.3 Minimal Inter-Core Communication. Initially, each thread was writing the result directly to the memory allocated by the main thread. Such operations are costly because they require many memory transfers across CPU cores. The new algorithms' implementations use a more efficient technique based on the OneTBB's task_scheduler_observer: each thread collects its partial result locally, and once the parallel work is done, a single memory transfer is issued from each thread towards the main thread so that latencies are minimized, and the overall computation performed faster. This optimization allowed the batch-point-location algorithm to postpone the memory bottleneck issue to bigger input instances. Unfortunately, the same optimization was not applied to the query points processing phase because the available time was insufficient.

7.1 Batch-Point-Location Performance Degradation Rate

In Figure 18, the batch-point-location algorithm is benchmarked with a fixed-size set of planes and as the amount of query points is increased from $10k$ to $20k$. This test proves the theory in [4] and Equation 7 as the number of query points affects way-less the algorithm's performance compared to the number of input planes H. For the sequential performances in Figure 14, every 15 planes added to H introduce a runtime penalty of $\approx$ 1.86x. On the other hand, sequential performances are penalized by $\approx$ 0.98x for every 10^3 additional query points added to Q.

7.2 Complexity And Running Time Comparisons

In Figures 19 and 20, the algorithms' runtimes are divided by the theoretical asymptotic complexities for the brute-force and batch-point-location algorithms. These tests provide a rough estimate of how well the implementation performs compared to the theoretical

algorithm. In the sequential case, the brute-force algorithm factors rapidly decrease below 1, approaching the value ≈ 0.2 for $|H| = 150$. This means that the algorithm performs within the theoretical upper bound and with a small constant factor; that is, the algorithm's implementation is efficient. This result matches our expectations, as the algorithm already showed a performance scaling factor in Figure 18. Regarding the batch-point-location algorithm, the estimated factors decrease quickly to ≈ 0.1 for $|H| = 130$ in the sequential case. Also, the constant factors of parallel implementations are bigger than those of sequential implementations. This is expected due to the I/O bottleneck and the overall performance penalty as the input size increases, as shown in 17. A similar trend can be seen for the brute-force algorithm: parallel versions using more cores are above all the other versions. The only exception applies to the 14 cores parallel implementation: the constant factors are smaller than the ones for the 13 and 12 cores versions. We do not know precisely what may cause this phenomenon, and more analysis is required. Both Figures have some line intersections, especially for very small input sizes. This is because the speed-up achieved with an additional processor rarely reaches 100%, and thus, dividing by the same number of processors may cancel out that advantage.

8 CONCLUSION

In this report, I explored some of the algorithms for the computation of three-dimensional lower envelopes described in [4] and [1]. First, a brief theoretical background and motivation for this project is provided. Next, a theoretical analysis of the algorithms is carried out: fundamental limitations, edge-cases and respective solutions are provided by means of code-snippets and descriptions. Finally, sequential and parallel benchmark results are discussed for the brute-force and batch point location algorithms. This project shows that parallel implementations of these algorithms are practically feasible and their theoretical complexities match the resulting running times. Nonetheless, many questions remain open and further work is required: faster brute-force and especially batch-point-location implementations are surely achievable; the main algorithm still requires some additional work in order for it to work, and a theoretical run-time complexity for the single sampling scheme should be determined.

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] T. M. Chan. 1996. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry* 16, 4 (apr 1996), 361–368. https://doi.org/10.1007/BF02712873

[3] Gianmarco Picarella. 2023. 3D Lower Envelope Algorithms. https://github.com/gianmarcopicarella/ga-spgmt-uu.

[4] Frank Staals. 2023. Parallel 3D Lower Envelope/Convex Hull. *Technical Report* (2023).