UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Space-efficient algorithms for triangulating a set of points in a plane

RELATORE:
Chiar.mo Prof. Andrea Alberto PIETRACAPRINA

SUPERVISORE:
Ing. Francesco SILVESTRI

LAUREANDO:
Nicola BARALDO

8 ottobre 2013

Anno Accademico 2012-2013

UNIVERSITÀ DEGLI STUDI DI PADOVA

Abstract

Facoltà di ingegneria
Dipartimento di Ingegneria dell'Informazione

Nicola Baraldo

Triangulating a point set in a plane is a basic computational geometry problem. Moreover, in the last years there has been an increment of interest in space-efficient algorithms. In this thesis we study some algorithms for the point set triangulation that take only $O(s)$ extra storage cells as workspace for any $1 \leq s \leq n$, where s is a parameter given as input. Our main results are an optimal sequential algorithm which takes $O(n(n/s + \log s))$ time, and a parallel algorithm which takes $O\left(\left(n^2 \log s\right) / (s \cdot p)\right)$, where p is the number of processors available. The parallel algorithm works on our memory-constrained parallel computational model, which is based on the CREW PRAM model.

Acknowledgements

I would like to acknowledge my advisor, Prof. Andrea Alberto Pietracaprina, for supporting me during my bachelor and master thesis. I also want to express my gratitude to Ing. Francesco Silvestri for his continuous help and support during my master thesis.

Above all, I want to thank my parents and my family for the continuous support and for giving me the opportunity to study during these years. Then I want to thank my old friends Andrea, Alberto and Giacomo who always encouraged me when any problem occured in my life. Moreover, I want to thank my University classmates and friends Matteo, Lorenzo and Denis, with whom I have shared many moments and expreriences. Last but not least, I want to express my deep appreciation to Beatrice for supporting me during this last year and for helping me with my English.

Contents

Dedicated to my parents.

Chapter 1

Introduction

1.1 Problems and motivations

In the last years we have seen a great growth of the amount of data to compute. In many cases, classical algorithms require additional space that can be as large as the size of the entire input. Depending on the size of the work-space required, we can have three different problems:

- the algorithm is not cache friendly because we have to work on data stored in distant positions in the main memory
- the work-space required by the algorithm cannot be entirely stored in the main memory, hence we have to use a secondary, slower memory so that it can kill the efficiency of the algorithm
- the algorithm cannot be used because we do note have enough memory to store the work-space (we remind the proliferation of tiny devices with limited memory)

These observations lead to an increasing interest in the memory requirements of computational tasks. Another relevant change in these last years is due to the flash memory, a new type of fast memory. One of the main feature of this memory is that the write operations are slow and expensive and they reduce the memory lifetime, while the read operations are fast and they do not afflict the memory lifetime such as the write operations. Under this situation, it makes sense to focus on algorithms that work with limited work-space, where the input is given in a read-only storage. We measure an algorithm's space efficiency by the number of work storage cells it uses.

The ultimate space efficiency is given by *constant-work-space* algorithms, which are the algorithms that use only a constant amount of variables in addition to the input storage. These algorithms are also said to run in log-space [2]. However, in this thesis we want to look for space-time trade-off algorithms, which means that we devise algorithms that are allowed to use up to $O(s)$ additional variables, for any parameter $1 \leq s \leq n$. Clearly, our aim is also to keep the running time of the algorithm as small as possible; in addition to this, we would like it to decrease while s grows.

In this thesis we want to focus also on memory-constrained algorithm on parallel computational models. In our case we shall study the *shared memory* model, also known as *parallel random access machine* PRAM model [18], which is one of the main parallel computational model. In particular, we consider the *concurrent-read exclusive-write* PRAM model (CREW PRAM). In the CREW PRAM model we have a memory where every processor can read and write; the read operations can be done concurrently to a single memory location, whereas the write operations are not allowed to a single memory location. In this thesis, we consider a situation where we have two memories:

- a limited *shared memory* where every processor can read and write, according to the CREW policy
- a *main memory* where every processor can only read and where the input resides

See Figure 1.1. This parallel memory-constrained model has gained lot of attention during the last years because it is compatible with modern parallel architectures. For example, one of these architectures is the GPU, *graphics processing units*; in this architecture we have a graphics card connected to a computer: the graphics card represents the PRAM architecture (processors and shared memory), while the computer's RAM represents the main memory. We will give a formal definition of this model later.

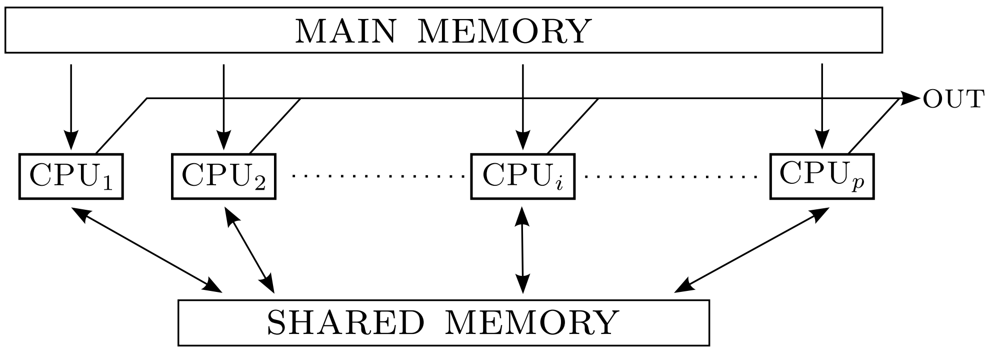


Figure 1.1: A graphical reprentation of our parallel computational model. The arrows represent the directions of the data movements.

Problem definition Under this situation we want to study a basic geometric problem like the triangulation of a point set on the plane. Let us give a more formal definition of this problem. Given a set P of n points in the plane, $P = \langle (x_1, y_1) \cdots (x_n, y_n) \rangle$, where every point lies on a different vertical line, join them by non intersecting line segments such that every region to the convex hull is a triangle.

1.2 Related results

One of the first memory-constrained algorithm was developed by Murno and Paterson [21] for the selection problem from an unsorted array. After this result, there have been more interests about these type of algorithms: during the years many algorithms were developed and many lower bounds were proved for different memory-constrained computational models [22, 10, 13].

In the last years two of the most famous memory-constrained computational models is the *streaming* model, where the values of the input can only be read once or a fixed number of times. On the streaming model, Chan and Chen [14] derive some algorithms for convex hull and linear programming. They obtain an algorithm for computing the convex hull of a n-point set in $O(n(\log s + n/s))$ time and $O(s)$ space on the multi-pass algorithm, for any parameter $s \leq n$.

Moreover, before this interest about memory-constrained algorithms, space-efficiency algorithms were represented by *in-place* algorithms. Such algorithms are allowed to permute the input instead of streaming or multi-pass algorithm. Therefore, the in-place model is a less restrictive model. Many geometric problems can be optimally solved with in-place algorithm [11, 12]. However, the main problem of in-place algorithm is that they tend to be complicated, they lack locality of reference and so they are less practical.

In between the in-place model and the multi-pass (or streaming) model we have another memory-constrained model, which is the *read-only* memory. In this model the input resides on a read-only memory, therefore we cannot permute the input. Since the input cannot be modified, the input does not need to reside in one place, as long as we can answer queries to access any individual element. Due to this property, the read-only model can be combined with external-memory models.

For what concerns the memory-constrained parallel model, we do not have results about computation geometry problems. There is an algorithm for the optimal sequence alignment in efficient space based on the message passing model (another parallel computational model). This algorithm runs in $O\left((m+n)^2/p\right)$ time and $O((m+p)/p)$ space per processor, where m and n are the size of the sequences and p is the number of processors available [17]. Another result regards time and space bounds for the implementation of the parallel programming language NESL on various parallel computational models, among which there is the concurrent-write and concurrent-read PRAM model (CRCW). They show that a NESL program with w work, d depth, and s space can be implemented on a p CRCW PRAM using $O(w/p + d\log p)$ time and $O(s + dp\log p)$ space [8].

For what concerns the triangulation problem, we remind that it is a well know problem and optimally solved in:

- $\Theta(n\log n)$ time on the classical serial computational model [23, 2]
- $\Theta(\log n)$ time on the CREW PRAM mode with n processors [20]

The main problem of these algorithms it that they both require $O(n)$ work-space. Recently, Asano et al. [4] developed some constant work-space algorithms for the point set triangulation and the point set Delaunay triangulation, which they run in $O(n^2)$ time. Regarding the polygon triangulation, which is a similar problem to the point set triangulation when the points stored on the input are sorted, we have another algorithm of Asano et al. [3]. Another recent result is given by Barba et al. [7]: they introduce a compressed stack technique that allows to transform stack-based algorithm into memory-constrained algorithm. Using this technique they solved many geometric problems, such as the polygon triangulation and the shortest path, which they run in $O(n^2/2^s)$ time and $O(s)$ space for any $s \in O(\log n)$ or $O((n\log n)/s)$ time and $O((p\log n)/s)$ space for any $2 \leq s \leq n$.

1.3 Computational models used

Now we shall discuss a more formal definition about the computational models used. In this thesis adopt two memory-constrained models: the first is a serial model and the second is a parallel one.

We suppose to have two function $x : P \to \mathbb{R}^2$ and $y : P \to \mathbb{R}^2$ such that, given a point $p \in P$, they return their abscissa and ordinate, respectively.

The input P is stored on a random access memory, where each element takes $O(\log n)$ bits. We measure the space efficiency of an algorithm by the number of work storage cells it uses, where a *cell* is a memory area that takes $O(\log n)$ bits; these cells are used as indices to the points of P, or they can contain a copy of an element of P.

Serial computational model Let s be a parameter, $1 \leq s$. The input P is stored in a read-only memory and the work-space available is limited to $O(s)$ extra storage cells; constant-time random access to the data is possible. Moreover, every basic arithmetic operation takes constant time. Our cost model is represented by the number of comparisons between the coordinates of the points. We assume that comparing the coordinates of the points takes constant time. The output is sent to a write-only output stream, hence when we write on the output-stream we must be certain that what we write is part of the final solution. To write in the output-stream we suppose to have the instruction **report**.

Parallel computational model Let p and s be two parameters, $1 \leq p \leq s$. We consider a computational model based on the *Concurred-Read, Exclusive-Write Parallel RAM* (CREW PRAM) model with p processors. The input P is stored on a read-only memory and the work-space available is limited to $O(1)$ storage cells per processor and $O(s)$ storage cells of shared memory that every processor can use (for example for communicate). Therefore, the total space available is $O(p + s) = O(s)$.

Our cost model is the same as the parallel complexity used in the PRAM model [18], which is the length of the critical path of the directed acyclic graph associated with any algorithm. We assume that comparing the coordinates of the points takes constant time, therefore each comparing operation can be represented by a single node of the directed acyclic graph.

The output is sent to a write-only stream, hence when we write on the output-stream we must be certain that what we write is part of the final solution. We suppose that every processor can concurrently write to the output stream without any loss of performance. To write in the output-stream we suppose to have the instruction **report**. Moreover, we suppose that the read-only memory is a slow memory whereas the shared memory is a fast memory. For modeling this situation we also count the number of read and write operations (we account them as if they were of the same type) that our algorithm makes. Let $W(n, s, p)$ be the number of read an write operations on the slow memory and let $V(n, s, p)$ be the number of read an write operations on the fast memory. In the end, to simplify the pseudocode of the algorithms we suppose that p is a power of two, s is a multiple of p and n is multiple of s.

Through the thesis, assume that the input is stored in general position: no points have the same $x-$ or $y-$coordinates; and no three points are collinear. For each of our algorithms is straightforward to remove this assumption.

1.4 Our results

In this thesis we present two algorithms for computing a point set triangulation:

- an optimal $O(n/s)$-pass algorithm that runs in

$$O\left(n\left(s \log s + \frac{n}{s}\right)\right)$$

time and takes $O(s)$ space, for any $1 \leq s \leq n$

- a parallel algorithm that runs in

$$O\left(\frac{n^2 \log s}{p \cdot s}\right)$$

time and takes $O(s)$ space on the CREW PRAM model with p processors, for any $1 \leq p \leq s \leq n$

For what concerns the second result, we want to point out some relevant cases with particular values of p and s:

- if $p = 1$ and $s = O(1)$ we obtain a serial constant work-space algorithm that runs in $O\left(n^2\right)$ time, which is the same result obtained by Asano et al. [4]
- if $p = 1$ and $s = O(n)$ we obtain a classic serial algorithm that runs in $O(n \log n)$ space and takes $O(n)$ space as the algorithms presented in [23, 2]
- if $p = \Theta(n)$ and $s = \Theta(n)$ we obtain a parallel algorithm that runs on the CREW PRAM model that takes $O(\log n)$ time and $O(n)$ space, which is the same result of Ed Merks [20]

1.5 Organization

Now we shall give a briefly description how this thiesis is organized.

Chapter 2 We present two serial algorithms: the first is the constant work-space algorithm of Asano et Al [4], the second is a general algorithm that takes $O(s)$ space without improving the first algorithm

Chapter 3 We improve the results in Chapter 2 by giving an optimal algorithm.

Chapter 4 Bulding on the optimal serial algorithm in Chapter 3, we derive a parallel algorithm that works on our parallel computational model.

Chapter 2

A simple triangulation algorithm

In this chapter we shall describe two space efficient algorithms for the point set triangulation that work on the serial computational model. The first one is the constant work space algorithm of Asano ed Al. [4]; it uses the *plane-sweep* approach, which is a basic computational geometry technique for developing algorithms. The second one is a general algorithm that is based on the same idea of the first one; it uses the *multi-pass* technique, which is a basic technique for developing space efficient algorithms.

First of all, let us define some notations that will be used during the chapter.

Definition 2.1. Let P be a point set of n points, then $\mathrm{T}(P)$ is the set containing every triangle of the triangulation of P.

Definition 2.2. Let P be a point set of n points, then $\mathrm{CH}(P)$ is the sequence of vertices of the convex hull of P in clockwise order.

Definition 2.3. Let P be a point set of n points, then $\mathrm{UH}(P)$ is the sequence of vertices of the upper hull of P in clockwise order starting from the leftmost point of P.

Definition 2.4. Let P be a point set of n points, then $\mathrm{LH}(P)$ is the sequence of vertices of the lower hull of P in clockwise order.

We want to note that if $\mathrm{UH}(P) = \langle r_1, \cdots, r_k \rangle$ and $\mathrm{LH}(P) = \langle u_1, \cdots, u_h \rangle$, then we have $\mathrm{CH}(P) = \langle r_1, \cdots, r_k, u_h, \cdots, u_1 \rangle$.

2.1 Constant work-space algorithm

In this section we want to present the constant work-space algorithm described in [4]. This model coincides with our model by setting $s = O(1)$.

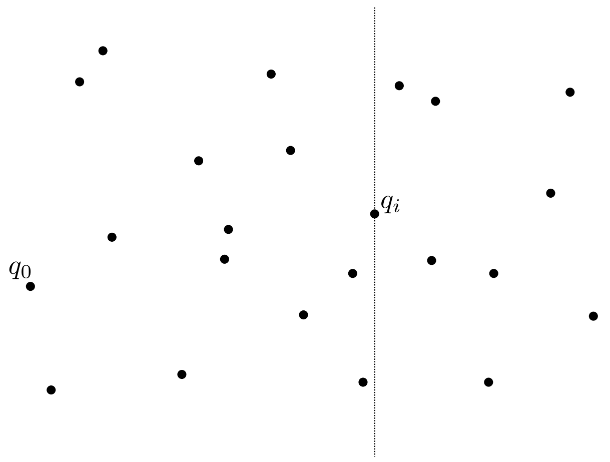


Figure 2.1: Plane sweep approach. It represents the input and the point q_i considered during the ith iteration.

Algorithm We use the sweep plane technique: the idea is to scan the points in nondecreasing order of x-coordinate adding each point to the triangulation of the points already considered. Let $\langle q_1 \cdots q_n \rangle$ be the sequence of the points in x-order. During the basic step we consider the point q_i, we compute the partial convex hull of the points on his left $\langle q_1 \cdots q_{i-1} \rangle$ that are visible form q_i and then we report all the triangles spanned by q_i and the edges of the partial hull. Clearly we cannot sort the points using an in-place algorithm because the input is stored on a read only memory and we do not have enough extra space to store the sorted sequence. Hence, to scan the points in order, at each step of the algorithm we perform a linear scan of the points for finding the next point to compute q_i given the previous point q_{i-1}.

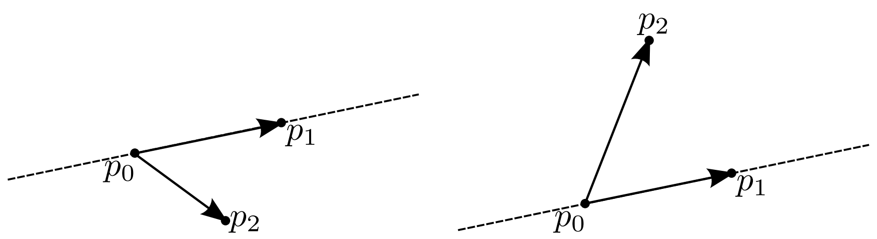
This algorithm can be done using only $O(1)$ extra work-space in this way:

1. find q_i with a linear scan of S
2. find the next upward hull edge e after q_{i-1} of the points $\langle q_1 \cdots q_{i-1} \rangle$
3. determine if e is visible from q_i using the previous convex hull edge discovered
4. if e is visible, report the triangle spanned by e and q_i and proceed to the next hull edge (step two). Otherwise stop and restart from q_{i-1} in the downward direction.

The pseudocode of the algorithm is given in Algorithm 1. NEXT_CLOCKWISE_-VERTEX finds the next upward hull vertex and NEXT_COUNTERCLOCKWISE_-VERTEX finds the next downward hull vertex, while VISIBLE determines if the edge e is visible from q_i. These functions can be done using $O(1)$ extra space, now we shall give the details of these functions.

Algorithm 1 Compute the triangulation of P using $O(1)$ space

```
function CONSTANT_SPACE_TRIANGULATION($P$)
    * find the three leftmost points of $P$: $q_1, q_2, q_3$ *
    report $\triangle(q_1, q_2, q_3)$
    for $i \leftarrow 4$ to $n$ do
        $q_i \leftarrow$ * leftmost point in $P$ to the right of $q_{i-1}$ *
        $u \leftarrow q_{i-1}$
        repeat
            $v \leftarrow$ NEXT_CLOCKWISE_NODE($P, q_i, u$)
            if VISIBLE($(u, v), q_i$) = TRUE then
                report $\triangle(q_i, u, v)$
                $u \leftarrow v$
            end if
        until VISIBLE($(u, v), q_i$) = FALSE
        $u \leftarrow q_{i-1}$
        repeat
            $v \leftarrow$ NEXT_COUNTERCLOCKWISE_NODE($P, q_i, u$)
            if VISIBLE($(u, v), q_i$) = TRUE then
                report $\triangle(q_i, u, v)$
                $u \leftarrow v$
            end if
        until VISIBLE($(u, v), q_i$) = FALSE
    end for
end function
```



(A) Clockwise turn. (B) Counterclockwise turn.

FIGURE 2.2: The dashed straight line that cross p_0 and p_1 divides the plane in two semiplanes. If p_2 lies below, $\overrightarrow{p_0p_2}$ makes a clockwise turn from $\overrightarrow{p_0p_1}$. If p_2 lies above, $\overrightarrow{p_0p_2}$ makes a counterclockwise turn from $\overrightarrow{p_0p_1}$.

Line segment geometry property Before showing how to implement the functions used in Algorithm 1, we show a basic property of line segments. Given two directed segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$, we show now how to determine if $\overrightarrow{p_0p_1}$ is clockwise from $\overrightarrow{p_0p_2}$ in respect to their common endpoint p_0. In [15], it is shown how to solve this problem in $O(1)$ time and space using the cross product. The cross product between two segments is:

$$\overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2} = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

The result is positive if $\overrightarrow{p_0p_1}$ is clockwise from $\overrightarrow{p_0p_2}$ (the points make a right turn)and it is negative if it is counterclockwise (the points make a left turn). In Figure 4.3 we can seen an example.

Property 2.5. *Let p_0, p_1 and p_2 be three points in the plane, then:*

1. *$\overrightarrow{p_0p_2}$ counterclockwise from $\overrightarrow{p_0p_1}$ if and only if p_2 lies on the left of the directed line from p_0 to p_1*
2. *$\overrightarrow{p_0p_2}$ clockwise from $\overrightarrow{p_0p_1}$ if and only if p_2 lies on the right of the directed line from p_0 to p_1*

Proof. The proof is trivial from the definitions of clockwise and counterclockwise turn (see Figure 4.3). Since no three points are collinear, these are the only possible cases. □

Corollary 2.6. *Let p_0, p_1 and p_2 be three points in the plane such that $x(p_0) < x(p_1)$, then:*

1. *$\overrightarrow{p_0p_2}$ counterclockwise from $\overrightarrow{p_0p_1}$ if and only if p_2 lies above the straight line that cross p_0 and p_1*
2. *$\overrightarrow{p_0p_2}$ clockwise from $\overrightarrow{p_0p_1}$ if and only if p_2 lies below the straight line that cross p_0 and p_1*

Find the next upward and downward hull vertex We use Jarvis' march algorithm, also known as gift wrapping algorithm [23, 15]. This algorithm allow us to find the next vertex on the convex hull in clockwise or in counterclockwise order of a point set respect to a point u that lies on the convex hull. Therefore, if u is the ith element in the sequence $\mathrm{CH}(P) = \langle r_1, \cdots, r_k \rangle$, $u = r_i$, then the next clockwise vertex after u is $r_{(i+1) \bmod k}$, and the next counterclockwise vertex after u is $r_{(i-1) \bmod k}$.

With a single scan of the point set, the algorithm finds the next vertex after u by comparing the polar angles of every point v respect u. If we want the next node in clockwise direction, it selects the point v' such that every other point lies on the right of the directed line from u to v'. Otherwise, if we want the next node in counterclockwise direction, it selects the point v'' such that every point lies on the left of the line from u to v''. During the computation we cannot find two points v'_1 and v'_2 that can both satisfy the condition just explained because there are not three collinear points existing.

From Property 2.5, to test if a point v lies on the left or on the right in respect to the directed line from u to v' is corrispondent to test if the directed segment $\overrightarrow{uv}$ is counterclockwise or clockwise from the directed segment $\overrightarrow{uv'}$.

Hence to find the next clockwise or counterclockwise hull node it takes $O(n)$ time and $O(1)$ space because we only have to scan the points and keep in memory the current node to return.

We remind that this algorithm should be executed only in a subset of the whole set P, subset that consists in the points that lie on the right of q_i, $\langle q_1 \cdots q_{i-1}\rangle$. Unfortunately we cannot store this subset, hence we have to scan the whole set P and consider only the points on the right of q_i. Refer to Algorithm 2 for the pseudocode of the clockwise version of the algorithm.

Algorithm 2 Find the next clockwise vertex after u in the convex hull of the points on the left of x

```
function NEXT_CLOCKWISE_VERTEX($P, x, u$)
    $n \leftarrow |P|$
    $* \; P = \langle p_1, \ldots, p_n \rangle \; *$
    $v \leftarrow *$ leftmost point in $P$ $*$
    for $i \leftarrow 1$ to $n$ do
        if ($x(p_i) < x$) then
            if ($\overrightarrow{uv} \times \overrightarrow{up_i} < 0$) then        ▷ if the cross product is negative
                $v \leftarrow p_i$
            end if
        end if
    end for
    return $v$
end function
```

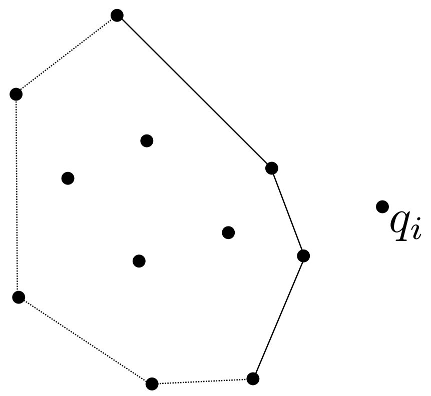


FIGURE 2.3: During the ith iteration we move along the convex hull of q_i. The solid segments are the segments of the convex hull of $\{q_1, \cdots, q_{i-1}\}$ that are visible from q_i. On the other hand, the dashed segments are the segments non visible from q_i.

Visibility Given an edge (u, v), $x(u) < x(v)$, that lies on the convex hull of $\langle q_1 \cdots q_{i-1} \rangle$, testing if this edge is visible from q_i is reduced to compute a cross product between two segments. Let r be the straight line that crosses u and v. If (u, v) is an edge of the upper hull of P, (u, v) is visible from q_i if and only if q_i lies above r. Otherwise if (u, v) is an edge of the upper hull of P, (u, v) is visible from q_i if and only if q_i lies below r. See Figure 2.3.

We test this property in this manner: if (u, v) is an upper hull edge, then (u, v) is visible from q_i if and only q_i is above r; on the other hand if (u, v) is a lower hull edge, then (u, v) is visible from q_i if and only if q_i is below r. From Property 2.6 and since $x(u) < x(v)$, the point q_i is above r if and only if the direct segment $\overrightarrow{uq_i}$ is counterclockwise from the direct segment $\overrightarrow{uv}$. Similary the point q_i is below r if and only if the direct segment $\overrightarrow{uq_i}$ is clockwise from the direct segment $\overrightarrow{uv}$.

This operation can be done in $O(1)$ time and space because testing if two segments make a clockwise or counterclockwise turn requires to compute the cross product between the endpoints of the directed segments [15]; see Algorithm 3 for details.

Algorithm 3 Return if the edge (u, v) is visible from p, where v lies on the convex hull

```
function VISIBLE($(u, v), p$)
    $c \leftarrow \overrightarrow{uv} \times \overrightarrow{uq_i}$                    ▷ cross product
    if $(u, v)$ is upper hull edge then
        return $c > 0$
    else
        return $c < 0$
    end if
end function
```

Theorem 2.7. *Given a set P of n points in $\mathbb{R}^2$, we can output the triangulation of P in $O(n^2)$ time and $O(1)$ extra space.*

Proof. We proceed by induction on the number of points n. If $n = 3$ there is only the triangle $\triangle(q_1, q_2, q_3)$ to report. Let us now suppose that the algorithm is correct for each $k < n$. For $n \geq 4$, let q_n be the rightmost point in P. Then by induction the algorithm correctly reports the triangulation of $P \setminus \{q_n\}$ and this triangulation lies inside the convex hull of $S \setminus \{q_n\}$. When q_n is considered, the algorithm reports all the possible triangles that connect q_n to the convex hull of $S \setminus \{q_n\}$. Moreover these triangles lie outside the convex hull of $S \setminus \{q_n\}$ because $x(q_{i-1}) < x(q_i)$, thus they are compatible with the triangulation already computed.

We shall show now the complexity of the algorithm. Finding the first triangle $\triangle(q_1, q_2, q_3)$ requires $O(n)$ steps. For each point q_i, $4 \leq i \leq n$, we have to find the previous point in order q_{i-1} and then compute the partial convex hull. Finding q_{i-1} requires a single scan of S, hence $O(n)$ steps. If k_i is the number of triangles reported during the ith iteration we have to execute k_i+2 Jarvis' march because we have to found also two points on the extremes that aren't visible from q_i. The Jarvis' march takes $O(n)$ steps, hence reporting these triangles costs $O(k_i n)$ time.

Since any set of n points contains at most $O(n)$ triangles [19], the total running time of this algorithm takes:

$$O(n) + \sum_{i=4}^{n} (O(n) + O(k_i n)) = O\left(n^2\right) + n \cdot O\left(\sum_{i=4}^{n} k_i\right) = O\left(n^2\right)$$

□

2.2 General algorithm

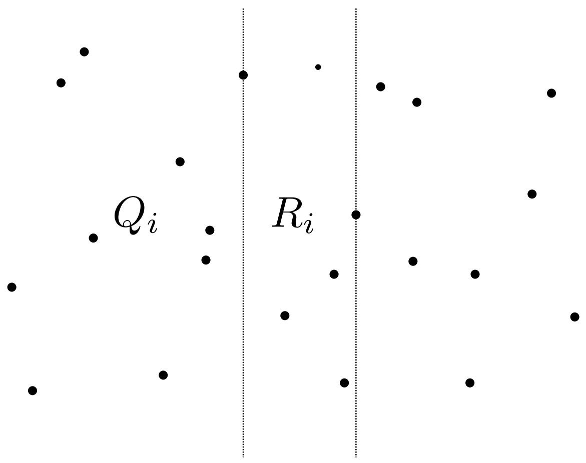


FIGURE 2.4: Multi pass approach. The input and the sets Q_i and R_i considered during the ith iteration. The set Q_i contains the points considered in the previous iterations. The set R_i contains the points currently considered.

Let us now suppose that we have $O(s)$ extra storage cells available. This algorithm is designed according to the multi pass approach. We consider the plane as partitioned in n/s vertical slabs, each one containing approximately s points. The algorithm scans the slabs from the left to the right: at each pass it computes the triangulation of the points inside the current slab σ and then merges the triangulation already computed to the triangulation of the points on his left. The pseudocode of this algorithm is given in Algorithm 4.

Algorithm RIGHT_SLAB (P, v, s) returns the next slab to compute, Algorithm SIMPLE_TRIANGULATION (R) returns the triangulation Q of R using

Algorithm 4 Compute the triangulation of P using $O(s)$ space

```
function MULTI_PASS_TRIANGULATION($P$)
    $v \leftarrow (-\infty, 0)$
    while $v \neq$ the rightmost point do
        $R \leftarrow$ RIGHT_SLAB($P, v, s$)
        $Q \leftarrow$ SIMPLE_TRIANGULATION($R$)
        for all $q \in Q$ do
            report $q$
        end for
        MERGE($P, v, R$)
        $v \leftarrow *$ rightmost point of $R$ $*$
    end while
end function
```

a classical algorithm for computing the triangulation [23, 2] and at the end Algorithm MERGE (P, v, R) computes all the triangles that connect the triangulation Q to the triangulation already computed at the left of R.

We shall show now how the functions RIGHT_SLAB (P, v, s) and MERGE (P, v, R) can be done.

2.2.1 Find the right slab

Algorithm RIGHT_SLAB(P, v, s) can be done in $O(n)$ time with a single scan of P using a buffer of size $2s$ and maintaining in the buffer the current s rightmost points at the left of v. We proceed in this way:

1. add the next s points and add them into the buffer, which initially contains s points
2. select the median of these $2s$ points
3. keep only the s smallest elements in the buffer and remove the others, then repeat from step one

Steps 1 and 3 can be done in $O(s)$ time, also step 2 requires $O(s)$ time using the function PICK (B', s) [9], which finds the sth smallest element in B' in linear time respecting to the size of B'. Moreover, each step requires $O(s)$ space because the buffer contains at most $2s$ points. The algorithm scans the points once; each iteration that this algorithm performs makes it advancing of s points, therefore there are at most $O(n/s)$ iterations. Since each iteration takes $O(s)$ time and space, the overall algorithm takes

$$O\left(\frac{n}{s} \cdot s\right) = O(n)$$

time and $O(s)$ space.

For a better understanding of this function see Algorithm 5. In that we have split the pseudocode in two functions where REMOVE_LAST(R, s) (Algorithm 6) is the function that removes the greatest elements (steps two and three).

Algorithm 5 Return the s leftmost points at the right of v in P

```
function RIGHT_SLAB($P, v, s$)
    $n \leftarrow |P|$
    $* \ P = \{p_1 \dots p_n\} \quad p_i = (x_i, y_i) \ \forall i \ *$
    $R \leftarrow \emptyset$
    $t \leftarrow 0$                          ▷ Represents the size of the buffer
    for $i \leftarrow 1$ to $n$ do
        if $x_i > v$ then
            $R \leftarrow \cup \{p_i\}$
            $t \leftarrow t + 1$
            if $t = 2s$ then
                $R \leftarrow$ REMOVE_LAST(R,s)
                $t \leftarrow s$
            end if
        end if
    end for
    if $t \geq s$ then
        $R \leftarrow$ REMOVE_LAST(R,s)
    end if
    return $R$
end function
```

Algorithm 6 Keep only the s smallest point and remove the others from R

```
function REMOVE_LAST($R, s$)
    $R' \leftarrow R$
    $\bar{x} \leftarrow$ PICK($R', s$)          ▷ It finds the $s$th smallest point
    for all $(x, y) \in R$ do
        if $x > \bar{x}$ then
            $R' \leftarrow R \setminus \{(x, y)\}$
        end if
    end for
    return $R'$
end function
```

2.2.2 Merge the current triangulation

In the constant work-space algorithm, for each point q_i we have computed all the triangles that can be formed having q_i as a vertex and the edges of the convex hull of the points $q_1 \dots q_{i-1}$. We have done this by scanning the convex hull using the Jarvis' march first in upward and then in downward direction. The merging phase can be done using the same idea with the exception that in this case we have s points to merge.

Problem Let Q_i be the set which contains the points on the left of the current slab σ, and let R_i be the set which contains the points inside σ during the ith iteration. We want to report every triangle that connects the triangulation of Q_i to the triangulation of R_i.

Algorithm During the analysis of the constant work-space algorithm we have noticed that, given a point set Q_i and a point p outside the convex hull of Q_i, we can add p into the triangulation of Q_i by reporting every triangle spanned by p and every edge on the convex hull of Q_i that is visible from p.

Let $Q'_i \subseteq \mathrm{CH}(Q_i)$ be the portion of the convex hull of Q_i that is visible from R_i, and $R'_i \subseteq \mathrm{CH}(R_i)$ be the portion of the convex hull of R_i that is visible from Q_i; let t_i^u and t_i^l be the upper and the lower common tanget of Q_i and R_i. Note that by the definition of upper and lower common tangent [23] the segments t_i^u and t_i^l connect the convex hull of Q_i and the convex hull of R_i. We consider the polygon U_i formed by the two partial hulls Q'_i, R'_i and the two segments t_i^u, t_i^l.

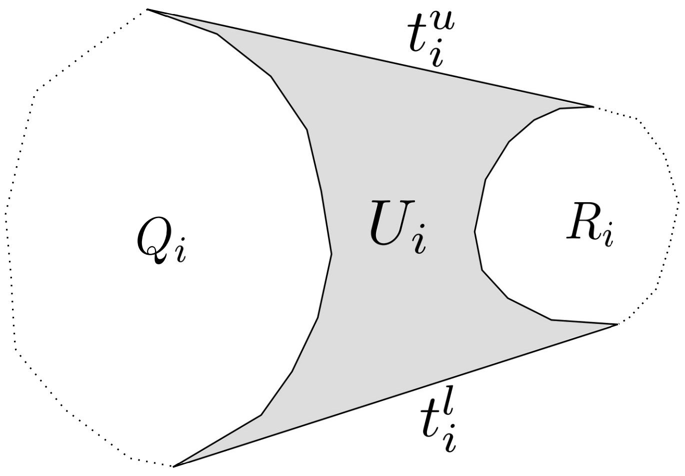


Figure 2.5: Merge phase: Q_i is the set of points alread computed, R_i is the set of points involved in the current iteration and the solid line defines U_i. In grey we can see the inner area of U_i.

We remind that $\mathrm{T}(X)$ represents the triangulation of a point set X. For the merge phase we have the following lemma:

Lemma 2.8. *Let Q_i and R_i be two point sets such that there exists a vertical line which divides these sets in two dinstict semiplanes, then $T(S_i \cup R_i)$ is the union of $T(Q_i)$, $T(R_i)$ and the triangulation of the polygon U_i.*

Proof. Since there exists a vertical line that divides Q_i and R_i we have that the convex hull of $Q_i \cup R_i$ is split in three parts: the points inside the convex hull of Q_i, the points inside the convex hull of R_i and the polygon U_i. The triangulations $\mathrm{T}(Q_i)$ and $\mathrm{T}(R_i)$ are already computed and reported, thus the remaining triangles to report are the ones inside the triangulation of the polygon U_i. □

We can apply Lemma 2.8 to Q_i and R_i, hence the merging phase is reduced to compute the triangulation of the polygon between $\mathrm{CH}(Q_i)$ and $\mathrm{CH}(R_i)$ limited by the upper and lower tangents of Q_i and R_i.

On our model we have at most $O(s)$ storage cells available, hence we have enough memory to store $\mathrm{CH}(R_i)$ because $|R_i| = s$, but not enough to store $\mathrm{CH}(Q_i)$. This means that also in this case we have to use the Jarvis' march to scan the edges of the convex hull of P_i without storing them as we have done in the constant work space algorithm.

We can report every triangle with the same technique adopted for computing the upper and lower tangents for the two disjoint polygons Q_i and R_i used in the divide and conquer convex hull algorithm [23]. Let $q \in Q_i$ and $r \in R_i$ be respectively the rightmost and leftmost points in Q_i and R_i. We use a pair (q', r'), $q' \in \mathrm{CH}(Q_i)$ and $r' \in \mathrm{CH}(R_i)$, which represents the current edge between $\mathrm{CH}(Q_i)$ and $\mathrm{CH}(R_i)$. Starting with $(q', r') = (q, r)$ at each step of the algorithm we first move q' or r' in the upward direction reporting every triangle found until (q', r') is the upper tangent of $\mathrm{CH}(Q_i)$ and $\mathrm{CH}(R_i)$ (the segment t_i^u in Figure 2.5). Then we repeat the same procedure in the downward direction until the lower tangent of $\mathrm{CH}(Q_i)$ and $\mathrm{CH}(R_i)$ is reached. The pair (q', r') is updated in this way:

1. using the Jarvis' march starting from q' find the next hull edge e of Q_i, if e is visible from r' report the triangle spanned by e and update q' to the next hull node (note that when e is not visible q' remains the same)

2. when the first edge e not visible from r' is found go to the next upward edge of $\mathrm{CH}(R_i)$, f

3. if f is visible from q' report the triangle spanned by f and q' and repeat from step one, otherwise stop in this direction and repeat in the downward direction from (q, r) in the same way

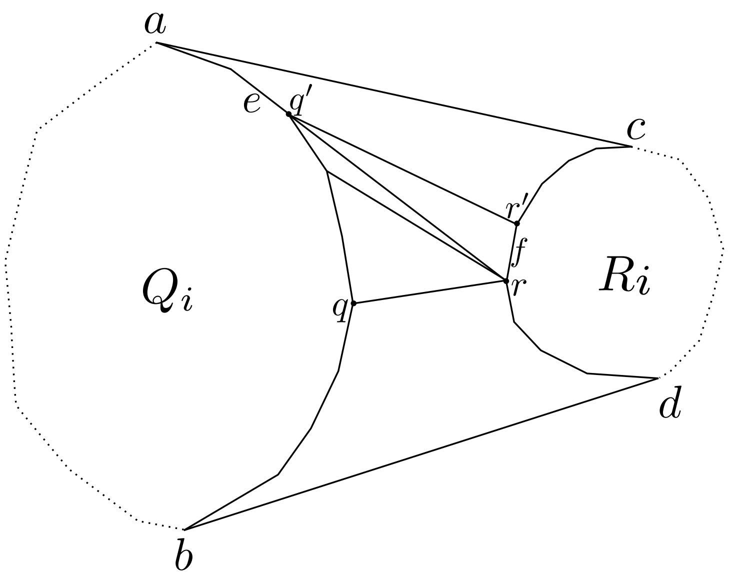


FIGURE 2.6: First iteration of the three steps

When the edge f is not visible from q' we have that also e is not visible from r', hence (q', r') is the upper (lower) tangent of P_i and R. The pseudocode of this function is given in 7.

In Figure 2.6 we can see the first time when the three steps are executed. Initially $(q', r') = (q, r)$, during the first step q' is updated in the upward direction along the convex hull of Q_i until the first node is visible from r. During the second step we can see the first edge non visible from r, e , and the next upward hull edge of R_i after r, f. In this case f is visible from q', hence in the third step the triangle spanned by f and q' is reported and we repeat the process from the first step.

Analysis At each iteration of the first step at least q' or r' are moved, hence at each iteration the edge (q', r') is moved towards the tangent of P_i and R (upward or downward, it depends on the direction). Moreover $|\mathrm{CH}(Q_i)| = O(n)$ and $|R_i| = O(s)$, hence the merge phase terminates.

First we shall show the correctness of Algorithm MERGE. If $Q_i' \subseteq \mathrm{CH}(Q_i)$ is the portion of the convex hull of Q_i visible from R_i and $R_i' \subseteq \mathrm{CH}(R_i)$ is the portion of the convex hull of R_i visible from Q_i, from the Lemma 2.8 we have to prove only that the merge phase correctly reports every triangle in the polygon U_i.

We consider now the first phase of Algorithm MERGE, when the triangles of U_i above (q, r) are reported. We shall show that for each iteration of the algorithm every triangle in U_i between (q, r) and (q', r') is reported in the previous iterations. During each iteration we can report a triangle in the first or in the third step of the algorithm. In the first step we report every

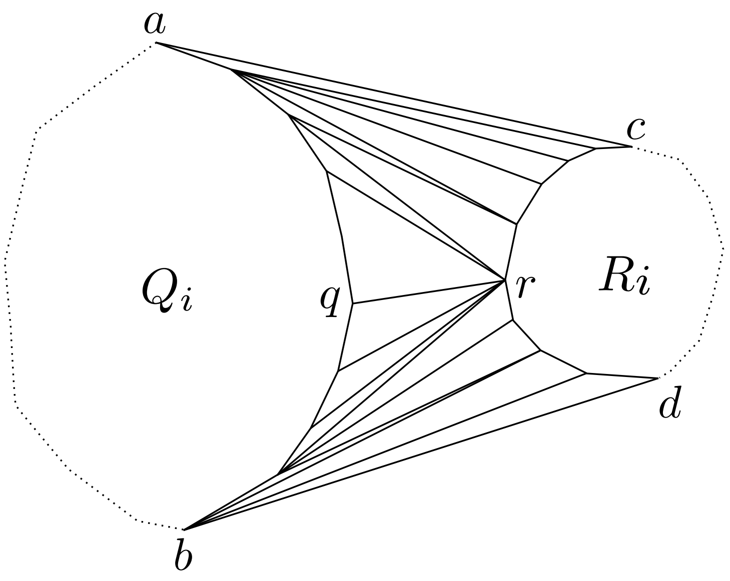


FIGURE 2.7: Merge phase

triangle above (q', r') with a vertex on r'; these triangles are all compatible with each other and with the triangles reported in the previous iterations as the lowest of them is adjacent to (q', r'). In the third step the only triangle reported is the one spanned by q' and the next upward hull edge after r', f. This triangle is reported only if f is visible from q' and it is adjacent to (q', r'). At the end of these steps the triangles reported can be merged into the triangles previously reported and (q', r') are updated according to the last triangles just reported. These considerations can be done during the second phase of Algorithm MERGE.

For what concerns the complexity we have to compute the upper hull and the lower hull of R_i first, then scan the portion of the convex hull of Q_i which is visible from R_i, Q_i'. Given a set of s points, computing the upper and the lower hull of this set takes $O(s \log s)$ time and $O(s)$ space [15, 2]. If k_i is the number of vertices of Q_i', then scanning Q_i' takes $O(k_i n)$ time and $O(1)$ space because we have a linear scan of P due to Jarvis's march algorithm. Therefore the overall cost of this algorithm is

$$O(s \log s + k_i n)$$

time and $O(s)$ space.

2.2.3 Total complexity

Algorithm 4 requires n/s passes. The cost of each pass is the sum of the three functions just explained, hence we have that the ith phase takes

$$O(n + k_i n + s \log s)$$

time and $O(s)$ space. Since $k_i \leq n$, the total cost required by the general algorithm is

$$\sum_{i=1}^{n/s} O(n + s \log s + k_i n) = O\left(\frac{n^3}{s} + n \log s\right) = O\left(\frac{n^3}{s}\right)$$

time and $O(s)$ space.

As we can see this algorithm is not better than the constant work-space algorithm previously explained. However we can see that the overall time taken by the general algorithm decreases as the parameter s increases. The main problem of this approach is that, during the ith phase of the general algorithm, Algorithm MERGE uses Jarvis's march algorithm in order to scan the convex hull of Q_i.

In the next session we will explain how to improve this algorithm in order to get a better result than the one obtained with the constant work-space algorithm.

Algorithm 7 Report every triangle that connects Q_i to R_i

```
function MERGE($P, v, R$)
    $R' \leftarrow$ SIMPLE_CONVEX_HULL($R$)
    $* \ R' = \langle r_1, \ldots, r_m \rangle \ *$
    $r \leftarrow *$ leftmost point in $R'$ $*$
    $q' \leftarrow v, \quad r' \leftarrow r$
    repeat
        $v \leftarrow$ NEXT_CLOCKWISE_NODE($P, q'$)
        $e \leftarrow (q', v)$
        if VISIBLE($e, r'$) = TRUE then
            report $\triangle$ $(r', q', v)$
            $q' \leftarrow v$
        else
            $u \leftarrow *$ node after $r'$ in $R'$ $*$
            $f \leftarrow (u, r')$
            if VISIBLE($f, r'$) = FALSE then          ▷ upper common tangent
reached
                break
            else
                report $\triangle$ $(u, r', q')$
                $r' \leftarrow u$
            end if
        end if
    until TRUE
    $q' \leftarrow v, \quad r' \leftarrow r$
    repeat
        $v \leftarrow$ NEXT_COUNTERCLOCKWISE_NODE($P, q'$)
        $e \leftarrow (q', v)$
        if VISIBLE($e, r'$) = TRUE then
            report $\triangle$ $(r', q', v)$
            $q' \leftarrow v$
        else
            $u \leftarrow *$ node before $r'$ in $R'$ $*$
            $f \leftarrow (u, r')$
            if VISIBLE($f, r'$) = FALSE then          ▷ lower common tangent
reached
                break
            else
                report $\triangle$ $(u, r', q')$
                $r' \leftarrow u$
            end if
        end if
    until TRUE
end function
```

Chapter 3

An optimal triangulation algorithm

In this chapter we shall explain an optimal algorithm for computing the convex hull on our serial computational model. This algorithm derives from the general algorithm in the previous chapter.

The main problem of the general algorithm is that for each reported triangle we have to pay the cost required by Jarvis' march algorithm, which takes $O(n)$ time and $O(1)$ space, hence we have that $O(s)$ space is wasted.

Before explaining how this algorithm works, we first introduce *Graham's scan* algorithm, one of the most famous algorithms in computational geometry.

3.1 Graham's scan

Graham's scan algorithm [23, 15] is an optimal time algorithm for computing the convex hull of a point set P under the classical computational model.

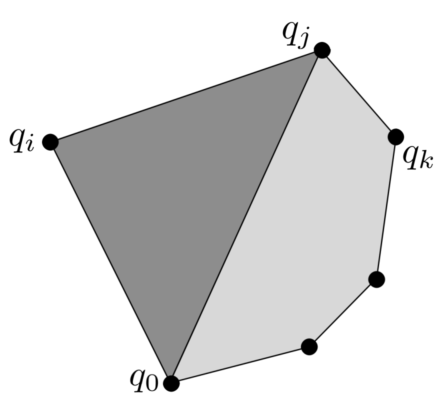
This algorithm uses an incremental approach, which is a very common technique in computational geometry. We use a data structure that represents the current convex hull of the already computed points. At each iteration of the algorithm we select a not yet considered point, then we update the data structure according to its position.

Graham's scan algorithm uses a stack S which represents the current convex hull of the already computed points in clockwise order.

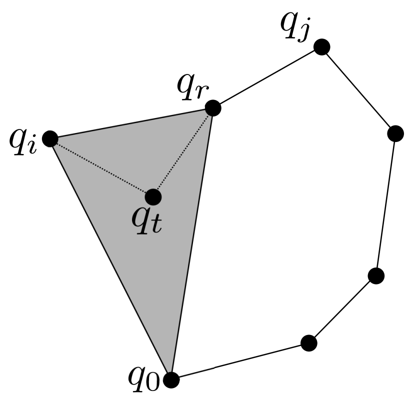
Algorithm We summarize the algorithm in these steps:

1. find the lowest point q_0 of P

2. sort the remaining points by polar angle in counterclockwise order around q_0 (if some points have the same polar angle, keep only the



(A) Adding the new point q_i into $\mathrm{CH}(\{q_0, \cdots, q_j\})$ gives the vertices of $\mathrm{CH}(\{q_0, \cdots, q_j, q_i\})$.



(B) The point q_t lies inside the triangle of vertices q_i, q_r and q_0. Thus it cannot be vertex of $\mathrm{CH}(\{q_0, \cdots, q_i\})$

FIGURE 3.1: The proof of correctness of the Graham's scan algorithm.

farthest point from q_0 and remove the others) and let $\langle q_1, \cdots, q_m\rangle$ be this sequence

3. let S be an empty stack and push the points q_0, q_1 and q_2 in this order
4. let s_1 and s_2 denote the top two points in stack S at any time
5. scan the sequence $\langle q_3, \cdots, q_m\rangle$ in order and do the following operations for each point q_i:
   (a) while the points q_i, s_1 and s_2 make a left turn, pop s_1 (the top point) from S
   (b) push q_i into S

Analysis We shall prove now the correctness of this algorithm. Let us prove by induction on the ith iteration of step 5 that, at the beginning of the ith iteration, stack S consists of the vertices of $\mathrm{CH}(\{q_0, \cdots, q_{i-1}\})$ in counterclockwise order from the bottom to the top.

We first note that after step 2 there are no three collinear points existing. If $i = 3$, stack S consists of the vertex $\{q_0, q_1, q_2\}$. These three points are not collinear, hence they form their own convex hull. By the ordering of the points, they appear in counterclockwise order from bottom to top.

Suppose now that this statement is valid for each hth iteration, $h < i$. If $i > 3$, let $s_1 = q_j$ be the top point on S after step 5a and $s_2 = q_k$ be the point just below q_j on S, clearly $k < j < i$. By the inductive hypothesis, after step 5a stack S contains the vertices of the convex hull of the set $\{q_0, \cdots, q_j\}$, $\mathrm{CH}(\{q_0, \cdots, q_j\})$, in counterclockwise from the bottom to the top because q_j is the top point in S. Moreover, we have that:

1. from the ordering of $\langle q_0, \cdots, q_i \rangle$, q_i's polar angle relative to q_0 is greater than q_j's polar angle

2. the points q_i, q_j and q_k make a right turn because otherwise they would have popped from S during step 5a

Therefore, since S contains the vertices of $\mathrm{CH}(\{q_0, \cdots, q_j\})$, after pushing q_i (step 5b) the stack S will contains the vertices of $\mathrm{CH}(\{q_0, \cdots, q_j\} \cup \{q_i\})$ (see Figure 3.1a to better understand this situation):

$$S = \mathrm{CH}(\{q_0, \cdots, q_j\} \cup \{q_i\}) \tag{3.1}$$

Let q_t be any point that was popped during step 5a and q_r the point right below q_t in stack S, then the points q_i, q_t and q_r make a left turn. Therefore we have that:

1. from Property 2.5, the point q_t lies on the right of the direct line that crosses q_i and q_r

2. in according to the positions where these points are located and the ordering of $\langle q_1, \cdots, q_i \rangle$, we have $1 < r < t < i$ and q_t lies inside the funnel defined by the two half lines $\overrightarrow{q_0 q_i}$ and $\overrightarrow{q_0 q_r}$

From these observation, the point q_t must lies in the interior of the triangle q_0, q_r and q_t (see Figure 3.1b). Hence q_t cannot be a vertex of $\mathrm{CH}(\{q_0, \cdots, q_i\})$, thus

$$\mathrm{CH}(\{q_0, \cdots, q_i\} - \{q_t\}) = \mathrm{CH}(\{q_0, \cdots, q_i\})$$

Let U_i be the set of points popped during the ith iteration of step 5, then we can apply the equality just proved for all the points in U_i, thus we obtain

$$\mathrm{CH}(\{q_0, \cdots, q_i\} - U_i) = \mathrm{CH}(\{q_0, \cdots, q_i\}) \tag{3.2}$$

But we have said that at the end of step 5a the point q_j is the head of the stack, $s_1 = q_j$, hence:

$$\{q_0, \cdots, q_i\} - U_i = \{q_0, \cdots, q_j\} \cup \{q_i\} \tag{3.3}$$

Putting together 3.1, 3.2 and 3.3 we obtain the following relationship:

$$S = \mathrm{CH}(\{q_0, \cdots, q_j\} \cup \{q_i\}) = \mathrm{CH}(\{q_0, \cdots, q_i\} - U_i) = \mathrm{CH}(\{q_0, \cdots, q_i\})$$

In the next iteration q_{i+1} is considered and the statement is proved.

At the end of step 5 we have $i = m + 1$, hence $\{q_1, \cdots, q_{i-1}\} = P$ and by the statement just proved we have $\mathrm{CH}(\{q_0, \cdots, q_{i-1}\}) = \mathrm{CH}(P)$.

We shall now show the time taken by this algorithm. The first and second steps require to find a minimum and to sort n points, hence these steps take $O(n \log n)$ time. Regarding step 5, the pop and push operation on stack S takes $O(1)$ time each. Every point in P is pushed exatly once (steps 3 and 5b). Moreover, at each stack S contains only points previously considered. Therefore, during step 5a when the point p_t is popped it will never be pushed because it can be pushed only once. The number of points is $O(n)$, thus the number of pop and push operations is $O(n)$. Since every pop and push operation costs $O(1)$, step 5 takes $O(n)$ time. Hence the overall costs of Graham's scan algorithm is $O(n \log n)$.

For what concerns the space requirements, the space needed by Graham's scan algorithm is the space for storing n sorted points plus the space needed to store stack S. Since set P contains n points, the number of points stored in stack after every iteration is $|S| = |\mathrm{CH}(\{q_0, \cdots, q_i\})|$ which is $\Theta(i)$ in the worst case. Therefore, at the end of the algorithm we shall have $|S| = |\mathrm{CH}(P)| = \Theta(n)$ in the worst case.

3.1.1 Another approach to the Graham's scan algorithm

The algorithm that we shall now present is a variant of the Graham's scan algorithm and it is also knwon as Andrew's monotone chain algorithm [1, 2]. Also in this case the algorithm uses an incremental approach. The main difference is that in this case the points P are not sorted by polar the angle but by the x-coordinate. As Graham's scan algorithm, this one mantanis a stack S that contains the vertices of the upper hull (or lower hull) of the points previously scanned.

Algorithm We summarize this algorithm in these steps:

1. sort the points by x-coordinate in decreasing order and let $\langle q_1, \cdots, q_n \rangle$ be this sequence
2. let S be an empty stack and push the points q_0 and q_1 in this order
3. let s_1 and s_2 denote the top two points in stack S at any times
4. scan the sequence $\langle q_2, \cdots, q_m \rangle$ in order and do the following operations for each point q_i:
   (a) while the points q_i, s_1 and s_2 make a left turn, pop s_1 (the top point) from S
   (b) push q_i into S

To compute the lower hull we perform the same operations. The only difference is that during the last step, we pop the top element from the stack S if the top two points in S and the point q_i currently considered form a right turn.

Analysis We shall prove now the correctness of this algorithm. We demonstrate only the correctness of the first part of the algorithm, the one that computes the upper hull of P; the second part can be done in the same way.

Let us prove by induction on the ith iteration of step 4 that, at the beginning of step 4, stack S consists of the vertices of $\mathrm{UH}(\{q_0, \cdots, q_{i-1}\})$ from the left to the right. If $i = 2$, stack S contains the points q_0 and q_1 that clearly form the upper hull of the set $\{q_0, q_1\}$.

Suppose now that this statement is valid for each hth iteration, $h < i$. If $i > 2$, we proceed on the same way of Graham's scan algorithm. The analysis is exactly the same, except from the moment when we show that a point q_t cannot be the vertex of $\mathrm{UH}(\{q_0, \cdots, q_i\})$. Let q_t be any point that was popped during step 4a and q_r be the point right below q_t in stack S, then the points q_i, q_t and q_r make a left turn. In according to the positions where these points are located in S, it results that $r < t < i$. By the ordering of $\langle q_0, \cdots, q_i\rangle$ we have $x(q_i) < x(q_t) < x(q_r)$; thus, since these points make a left turn, by Corollary 2.6 the point q_t must lie below the straight line that crosses q_i and q_r. Therefore, the point q_t cannot be a vertex of $\mathrm{UH}(\{q_0, \cdots, q_i\})$.

Regarding the complexity of this algorithm: as for Graham's scan algorithm, the fist two steps take $O(n \log n)$ time, while step 4 takes $O(n)$ time. Hence the overall cost is $O(n \log n)$.

Regarding the space required by this algorithm, also in this case we need the space for storing n sorted points plus the space required to store stack S. Therefore, as for Graham's scan algorithm, the space required is $\Theta(n)$.

3.2 Compute the partial hull in $O(s)$ space

We shall now show how to speed up the computation of the partial hull using an algorithm proposed in [14]. This paper proposes an algorithm for computing the convex hull of a point set under the same model using $O(s)$ work space. We only show how to compute the partial upper hull, as the lower hull works in the same way. The algorithm uses the same approach as Andrew's monotone chain algorithm.

This algorithm shall be used in Algorithm MERGE in order to scan the upper hull (and the the lower hull) of the point set Q_i instead of Algorithm NEXT_COUNTERCLOCKWISE_VERTEX (and then Algorithm NEXT_CLOCKWISE_VERTEX).

The algorithm that we shall present takes as input: a point set P, a point $v \in P$ and a parameter $1 \leq s \leq n$. Let P' be the point set defined in the following way

$$P' = \{p \in P : x(p) \geq x(v)\}$$

The output is the sequence of points $\langle q_1, \cdots, q_r \rangle$, such that it contains the vertices of the upper hull of the point set P' from the right to the left (in counterclockwise clockwise order). The sequence $\langle q_1, \cdots, q_r \rangle$ has the following properties:

1. the sequence contains at least one vertex, and at most s vertices, hence we have
$$1 \leq r \leq s \tag{3.4}$$
2. at least s points of P' are covered by this sequence, hence we have
$$\left| \left\{ p \in P' : x\left(q_r\right) \leq x\left(p\right) \leq x\left(q_1\right) \right\} \right| \geq s \tag{3.5}$$

Therefore, the sequence returned by this algorithm takes at most $O\left(s\right)$ storage cells, hence it can be explicitly stored.

Algorithm The algorithm operates in this way:

1. find the s rightmost points in P', and let Q be the set containing these points;
2. compute the upper hull of Q
3. adjust the upper hull of Q taking into account also the points on the left of every point in Q

Since we cannot explicitly store P', step 1 require to find the s rightmost points on the left of v (including v itself). Therefore, it can be done adopting the same approach used for finding the next slab in the multi-pass algorithm (Algorithm RIGHT_SLAB), except for the fact that we have to include also v. Step 2 can be done using a classic convex hull algorithm, for example that Graham's scan algorithm just explained. Step 3 is more complicated: we use the approach explained at the end of the previous section.

Let $\langle q_1, \cdots, q_m \rangle$ be the sequence of the vertices of the upper hull of Q from the right to the left. In order to adjust the upper hull of Q we use the approach of Andrew's monotone chain algorithm. Therefore we use a stack S as auxiliary data structure. In this case we already have the vertices of $\mathrm{UH}\left(Q\right)$ stored, hence we initiate the stack with the sequence $\langle q_1, \cdots, q_m \rangle$ from the bottom to the top (q_m is the top most elements in S).

Let w be the leftmost point in Q and let $U = \{u_1, \cdots, u_l\}$ be the set containing every point that lies on the left w. Note that $Q \cup U$ represents the set of the points on the left of v plus v, hence:

$$P' = Q \cup U$$

Step 3 performs a scan of U and for each point $p \in U$ we perform the following operations:

1. pop every node from stack S until the top two elements make a left turn with the point p (as in step 4 of the Andrew's monotone chain algorithm)

2. if point p has produced some pop operations, then we push p into the stack in order to have a segment that covers the points just popped, otherwise p cannot replace other points in the stack thus it is not pushed into S

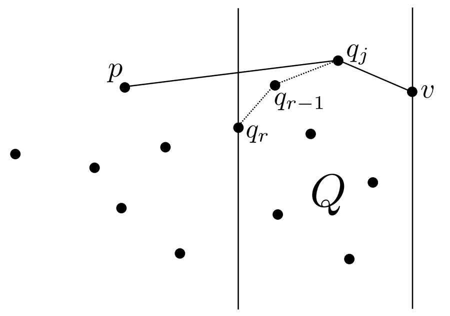
Note that for every push operation there is at least one pop operation, hence the size of the stack cannot increase. This means that we can implement stack S using the sequence $\langle q_1 \dots q_m \rangle$ plus a single variable r that represents a pointer to the head of the stack. A push operation corresponds to increment r, and a pop operation corresponds to decrement r. Since $|Q| = s$, we have $m = |\mathrm{UH}(Q)| = O(s)$, hence stack S takes $O(s)$ storage cells.

If LEFT_SLAB (P, v, s) is the function that returns the s rightmost points on the left of v and SIMPLE_UPPER_HULL (Q) is the function that compute the upper hull of Q. We can write the pseudocode of this function as follows on Algorithm 8.

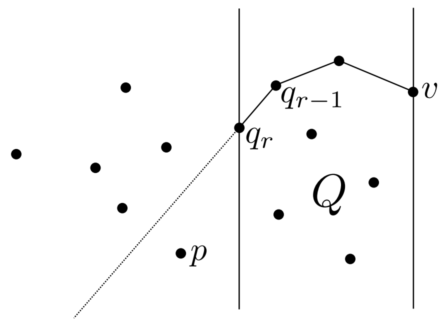
Algorithm 8 Compute a portion of the upper hull from v

```
function PARTIAL_UPPER_HULL($P, v, s$)
    $Q \leftarrow$ LEFT_SLAB($P, v, s$)
    $Q \leftarrow Q \cup \{v\}$
    $\langle q_1 \dots q_m \rangle \leftarrow$ SIMPLE_UPPER_HULL($Q$)
    $r \leftarrow m$
    for all point $p$ to the left of of $v$ do    ▷ adjust the current upper hull
        $j \leftarrow r$
        while $p$ is above $\overline{q_{j-1}q_j}$ do
            $j \leftarrow j - 1$
        end while
        if $j < r$ then ▷ if a pop operation occured, update the upper hull
            $r \leftarrow j + 1$
            $q_r \leftarrow p$
        end if
    end for
    return $\langle q_1 \dots q_r \rangle$
end function
```

Lemma 3.1. *Given a set P of n points in the plane, a point $v \in UH(P)$, and a parameter s, $1 \leq s \leq n$, Algorithm* PARTIAL_UPPER_HULL *returns the sequence of vertices of the upper hull of P from v in counterclockwise direction in $O(n + s \log s)$ time and $O(s)$ space, with $O(1)$ passes of P.*



(A) The points p, q_r and q_{r-1} make a left turn hence the point q_r cannot be the vertex of the upper hull of P. In this case we have the same behavior of Andrew's monotone chain algorithm.



(B) The points p, q_r and q_{r-1} make a right turn hence the point p lies below the line that crosses q_r and q_{r-1} (the dashed line).

FIGURE 3.2: The proof of correctness of Algorithm PARTIAL_UPPER_HULL.

Proof. We prove by induction on the points of $U = \{u_1, \cdots, u_l\}$ parsed that, at the beginning of the ith iteration of the for-loop, the sequence $\langle q_1 \dots q_r \rangle$ is the portion of the upper hull of the set of the points $Q \cup \{u_1, \cdots, u_{i-1}\}$ from the point v in counterclockwise direction. The set of points just mentioned consists in the points of Q and any point considered in the previously iterations of the for-loop. Therefore, after the for-loop the points parsed by the algorithm are the points in $Q \cup \{u_1, \cdots, u_l\} = Q \cup U$.

The base case consists of the points of the set Q. From the correctness of Algorithm SIMPLE_UPPER_HULL, the sequence $\langle q_0 \dots q_r \rangle$ represents the whole upper hull of Q.

Suppose now that our statement is correct for every hth iteration, $h < i$. Let u_i be the point considered in the ith iteration of the for-loop. In the analysis of Andrew's monotone chain algorithm, we have showed that every point popped from S cannot be the vertex of the upper hull in $Q \cup \{u_1, \cdots, u_{i-1}\}$.

If during the for-loop there have been some pop operations ($j < r$), the point u_i is pushed into the stack. In this case this algorithm works exactly as Andrew's monotone chain algorithm, hence we have that the sequence $\langle q_0 \dots q_j, u_i \rangle$ covers every point of Q. Moreover, we have that:

1. from the ordering of $\langle q_0 \dots q_r \rangle$ we have $x(q_r) < x(q_{r-1})$
2. the points u_i, q_r and q_{r-1} make a left turn (otherwise it would be $j = r$)

From Corollary 2.6, the point u_i lies above the straight line that crosses q_r and q_{r-1}. By the inductive hypothesis we can add that every point in

$Q \cup \{u_1, \cdots, u_{i-1}\}$ lies below the straight line that crosses q_r and q_{r-1}. Therefore, every point in $Q \cup \{u_1, \cdots, u_i\}$ lies below the straight line that crosses q_r and q_{r-1}. After pushing the point u_i, the new segment $\overline{u_i q_j}$ will replace the old segment $\overline{q_r q_{r-1}}$ (Figure 3.2a).

If the points u_i, q_r and q_{r-1} make a right turn, during the for-loop no pop operations occurred ($j = r$). In this case, from Corollary 2.6 the point u_i lies below the straight line that crosses q_r and q_{r-1} and from the inductive hypothesis every point in $Q \cup \{u_1, \cdots, u_i\}$ lies below this straight line (Figure 3.2b).

At the end of the for-loop we have scanned the entire set U, hence every point $p \in (U \cup Q)$ lies below $\langle q_1, \cdots, q_r\rangle$ or lies below the straight line that crosses q_r and q_{r-1}. Therefore the sequence $\langle q_1, \cdots, q_r\rangle$ is a portion of $\mathrm{UH}(Q \cup U)$.

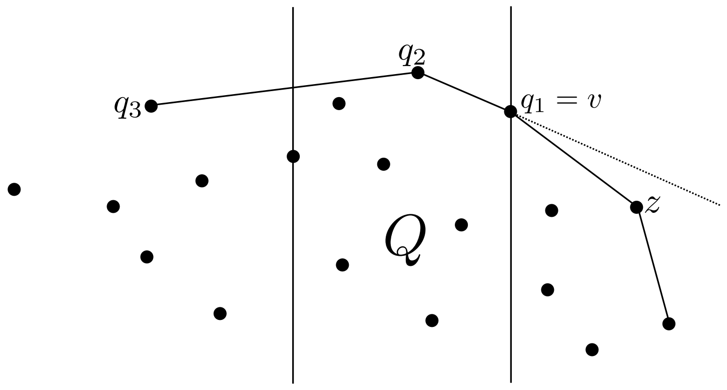


FIGURE 3.3: The points q_j, v and z make a non left turn, hence every point that lies on the right of v also lies below the straight line that crosses q_j and v (the dashed line).

Let us prove now that the sequence $\langle q_1, \cdots, q_r\rangle$ is also a portion of the upper hull of P. Since $v \in \mathrm{UH}(P)$, let z be the next vertex in clockwise order after v in $\mathrm{UH}(P)$ (Figure 3.3). Moreover, $v \in Q$ and it is the rightmost point in Q, thus we must have $q_1 = v$.

The vertices q_2, v and z make a right turn because otherwise v would not be in $\mathrm{UH}(P)$, and $x(q_2) < x(v) < x(z)$. Therefore, from Corollary 2.6, every point that lies on the right of v must be below the straight line that crosses v and z must be below the straight line that crosses q_1 and v. Moreover, since the sequence $\langle q_1, \cdots, q_r\rangle$ forms a chain that only makes right turns, every straight line that crosses two consecutive vertices of this sequence must covers every point on the right of v. Therefore, the sequence $\langle q_1, \cdots, q_r\rangle$ is part of the upper hull of P.

We shall prove now the complexity of this algorithm. The function SIMPLE_UPPER_HULL (σ) represents a simple convex hull algorithm that takes $O(s \log s)$ time and $O(s)$ space, for example Graham's scan algorithm. At the end adjusting the upper hull requires a simple scan of P which takes $O(n)$ time and $O(1)$ space. Hence the function PARTIAL_UPPER_HULL (P, v) computes at least one edge and at most $O(s)$ edges in $O(n + s\log s)$ time. Note that Algorithm RIGHT_SLAB and adjusting $\langle q_1 \dots q_j \rangle$ require only one scan of P each, hence Algorithm 8 takes only $O(1)$ passes of P. □

Let us now prove that the sequence returned by Algorithm PARTIAL_-UPPER_HULL satisfies Property 3.4 and 3.5.

Property 3.2. *The partial hull $\langle q_1 \dots q_r \rangle$ returned by Algorithm 8 on a set P covers at least s points of S.*

Proof. The sequence $\langle q_1 \dots q_r \rangle$ covers at least the points inside the set Q, which contains s points. □

Lemma 3.3. *Given a set P of n points in the plane and a parameter s, we can scan the vertices of the upper hull from the left to the right by a $O(n/s)$-pass algorithm which uses $O(s)$ space and runs in $O(n(n/s + \log s))$ time.*

Proof. From the Property 3.2, Algorithm 8 covers at least s points. Hence with at most $O(n/s)$ iterations of this algorithm, it covers the entire set P. Moreover from Lemma 3.1, Algorithm 8 requires only $O(1)$ passes of P, hence since there are $O(n/s)$ iterations we have $O(n/s)$ passes of P. □

3.3 Application to the triangulation algorithms

In this section we shall explain how to use Algorithm PARTIAL_UPPER_HULL instead of Algorithm NEXT_COUNTERCLOCKWISE_VERTEX in order to scan the set Q_i in Algorithm MERGE. Also in this case the scan of the lower hull of Q_i is similar.

We proceed in this way:

1. let q' be the current point considered in $\mathrm{UH}(Q_i)$;
2. use Algorithm PARTIAL_UPPER_HULL to compute the next points $\langle q_1 \dots q_r \rangle$ from q' in $\mathrm{UH}(Q_i)$;
3. report the next triangles using the same method until q' reaches q_r;
4. repeat from step two in order to obtain the next hull nodes.

Using this algorithm in the merge phase we have the following differences:

- Algorithm NEXT_COUNTERCLOCKWISE_VERTEX returns only the next vertex to compute instead of Algorithm PARTIAL_UPPER_HULL that may return more vertices;

- if the merge of Q_i and R_i requires to scan the entire convex hull of Q_i, from Lemma 3.3 using Algorithm PARTIAL_UPPER_HULL takes $O\left(n/s\left(n + s\log s\right)\right)$ steps, instead of Algorithm NEXT_COUNTERCLOCKWISE_-VERTEX which takes $O\left(n^2\right)$ time.

Let us define the following algorithms:

OPTIMAL_MERGE: Algorithm MERGE that uses Algorithm PARTIAL_UPPER_-HULL instead of Algorithm NEXT_COUNTERCLOCKWISE_VERTEX in order to scan the set Q_i

OPTIMAL_MULTI_PASS_TRIANGULATION: Algorithm MULTI_PASS_TRIANGULATION that uses Algorithm OPTIMAL_MERGE instead of Algorithm MERGE

3.4 Analysis

Now we want to analyse how Algorithm OPTIMAL_MULTI_PASS_TRIANGULATION works. Like we have seen in the first version of Algorithm MULTI_PASS_-TRIANGULATION, the main cost is due to the overall cost of the merge phases. Our aim is to provide a better bound to the overall time taken by the merge phases when using Algorithm OPTIMAL_MERGE instead of Algorithm MERGE.

This analysis is focused to understand how Algorithm PARTIAL_UPPER_-HULL works during each iteration, then we obtain an upper bound to the number of times that Algorithm PARTIAL_UPPER_HULL is called during the whole execution of Algorithm OPTIMAL_MULTI_PASS_TRIANGULATION. To simplify the analysis we consider only the portion of the upper hull scanned by Algorithm PARTIAL_UPPER_HULL. The analysis of the portion of the lower hull works on the same way and afflicts the total analysis only by a constant factor.

Definition 3.4. Given two segments $\overline{ab}$, $x\left(a\right) < x\left(b\right)$, and $\overline{cd}$, $x\left(c\right) < x\left(d\right)$, we say that $\left(a, b\right)$ *covers* $\left(c, d\right)$ if:

1. $\left(x\left(a\right) \leq x\left(c\right)\right) \wedge \left(x\left(d\right) \leq x\left(b\right)\right)$
2. c and d lies below the straight line that crosses a and b.

Property 3.5. *If* $\left(a, b\right)$ *covers* $\left(c, d\right)$ *and* $\left(c, d\right)$ *covers* $\left(e, f\right)$*, then* $\left(a, b\right)$ *covers* $\left(e, f\right)$*.*

Proof. Since $\left(a, b\right)$ covers $\left(c, d\right)$ and $\left(c, d\right)$ covers $\left(e, f\right)$ we have:

$$\left(x\left(a\right) \leq x\left(c\right)\right) \wedge \left(x\left(c\right) \leq x\left(e\right)\right) \Rightarrow x\left(a\right) \leq x\left(e\right)$$
$$\left(x\left(d\right) \leq x\left(b\right)\right) \wedge \left(x\left(f\right) \leq x\left(d\right)\right) \Rightarrow x\left(f\right) \leq x\left(b\right)$$

Let r_1 the straight line that crosses a and b and r_2 the straight line that crosses c and d. Since c and d lie below r_1 we have that every point in the segment $\overline{cd}$ lies below r_1, similary we have that also every point in the segment $\overline{ef}$ lies below r_2.

Consider the sets:

$$R_1 = \{p : x(a) \leq x(p) \leq x(b), p \text{ below } r_1\}$$
$$R_2 = \{p : x(c) \leq x(p) \leq x(d), p \text{ below } r_2\}$$

For each point p of R_2 we have that it lies below $\overline{cd}$, thus it lies also below r_1. Moreover, since $x\,(a) \leq x\,(c) \leq x(p) \leq x\,(d) \leq x\,(b)$ and p lies below r_1, we have that $p \in R_1$. Hence $R_2 \subseteq R_1$.

Since $e, f \in R_2$ and $R_2 \subseteq R_1$, also $e, f \in R_1$. Then e and f lies below r_1 and $(x\,(a) \leq x\,(e)) \wedge (x\,(f) \leq x\,(b))$, we have that the segment (a, b) covers the segment (e, f). $\square$

Let us consider the ith merge phase, Algorithm OPTIMAL_MERGE stops to scan the upper hull of Q_i only when the last triangle reported touches the upper common tangent of Q_i and R_i, segment t_i^u in Figure 2.5. Ater this observation we define the set which contains all these segments.

Definition 3.6. Let T_i^u be the set which contains all the upper common tangents for any phase of the algorithm:

$$T^u = \left\{t_1^u, \ldots, t_{n/s}^u\right\}$$

where t_i^u is the upper common tangent of Q_i and R_i at the ith phase.

Now we define another two sets which are the key of this analysis.

Definition 3.7. For each phase i, let $M_i^u \subseteq T^u$ the set containing the upper tangents involved in the merge phase as convex hull edges of Q_i.

Definition 3.8. For each phase i, let $N_i^u \subset Q_i$ the set of points below t_i^u but not below any $t \in M_i^u$.

In Figure 3.4 we can see the sets M_i^u and N_i^u during the ith phase. First we will study the relationship between the sets M_i^u and N_i^u among the merge phases. After proving this relationship, we will show how to bound the number of times Algorithm PARTIAL_UPPER_HULL is called during the ith merge phase as function of $|M_i^u|$ and $|N_i^u|$.

We start to prove the relationship between the sets $M_1^u, \cdots, M_{n/s}^u$ with the following lemma.

Lemma 3.9. *Consider the sets $M_1^u, \cdots, M_{n/s}^u$, then:*

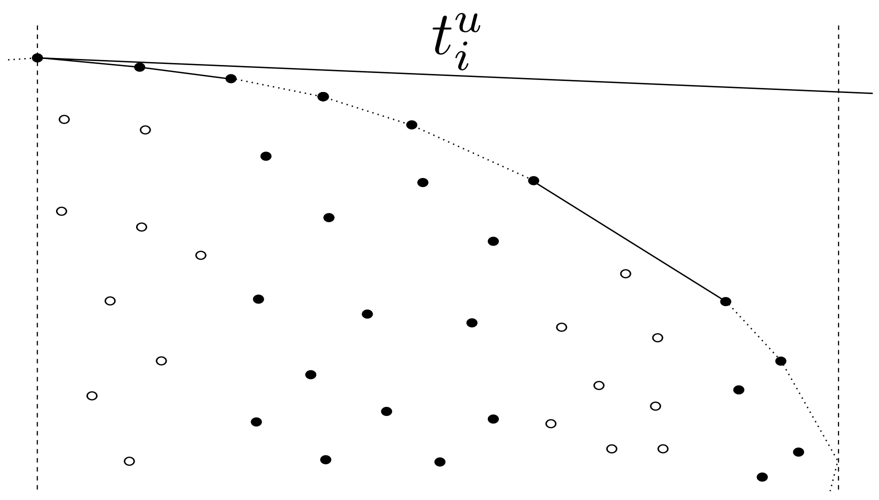


FIGURE 3.4: The continuous segments represent the set M_i^u and the circles are the points of Q_i covered by these segments. The dashed segments are upper hull edges of Q_i involved in the merge phase that were not tangent in the previous phases. The points below the dashed segments are the points of N_i^u. The segment that covers every points is the upper common tangent of Q_i and R_i, t_i^u.

1. $M_i^u \cap M_j^u = \emptyset \qquad \forall\ i, j,\ i \neq j$

2. $\sum_{i=1}^{n/s} |M_i^u| \leq n/s$

Proof. Let us focus on the ith phase and let be $t \in M_i^u$. From Definition 3.7, edge t is involved in the merge phase, hence Algorithm MERGE reports a triangle spanned by t and a vertex of R_i. This means that, at the end of the ith merge phase, edge t is an inner edge of the triangulation of $Q_i \cup R_i$. Therefore, t cannot be involved in the next merge phases, hence $t \notin \mathrm{CH}(Q_j)\ \forall\ j > i$. Therefore:

$$\forall\ t \in M_i^u,\ t \notin M_j^u \Rightarrow M_i^u \cap M_j^u = \emptyset \qquad \forall i < j$$

The first point is proved. The second point derive directly from the first point:

$$\sum_{i=1}^{n/s} |M_i^u| = \left| \bigcup_{i=1}^{n/s} M_i^u \right| \leq |T^u| = \frac{n}{s}$$

□

In order to prove a similar relationship between the sets $N_1^u, \cdots, N_{n/s}^u$, we have to show other relationships between the tangents $t_1^u, \cdots, t_{n/s}^u$ reported during the merge phases.

Lemma 3.10. *For each phase i and tangent $t \in \left\{t_1^u, \cdots, t_{i-1}^u\right\}$, there exists a tangent $t' \in \left\{t_1^u, \cdots, t_{i-1}^u\right\}$ such that $(t'$ covers $t) \wedge (t' \in UH(Q_i))$*

Proof. If $t \in \mathrm{UH}\,(Q_i)$ then t covers itself.

If $t \notin \mathrm{UH}\,(Q_i)$ (t lies inside $\mathrm{UH}\,(Q_i)$) then t is an internal edge of $T\,(Q_i)$ because every tangent edge belongs to a reported triangle (during a previous merge phase in this case). In particular consider the iteration k_1, $1 \leq k_1 < i$, where $t \in \mathrm{UH}\,(Q_{k_1})$ and it is involved in the merge phase ($t \in Q'_{k_1}$ according to our previous notation). During this iteration we have that t is involved in the merge phase and this is the iteration where t becomes an internal edge of the triangulation of Q_i. At the end of this merge phase $t_{k_1}^u$ covers t.

Now we have two possible cases: $t_{k_1}^u \in \mathrm{UH}\,(Q_i)$ or $t_{k_1}^u \notin \mathrm{UH}\,(Q_i)$. In the first case the proof is done. In the second case we repeat the same process and find the iteration k_2, $k_1 < k_2 < i$, such that $t_{k_2}^u$ covers $t_{k_1}^u$. Moreover if $t_{k_2}^u$ covers $t_{k_1}^u$ also $t_{k_2}^u$ covers t due to Property 3.5. If $t_{k_2}^u \in \mathrm{UH}\,(Q_i)$ we have done, otherwise we repeat this process more times until we find the iteration k_m, $k_1 < k_2 < \cdots < k_m < i$, such that $t_{k_m}^u \in \mathrm{UH}\,(Q_i)$. Since $t_{k_j}^u$ covers $t_{k_{j-1}}^u$ $\forall\ 1 < j \leq m$ and $t_{k_1}^u$ covers t, from Property 3.5 we have that $t_{k_m}^u = t'$ covers t. □

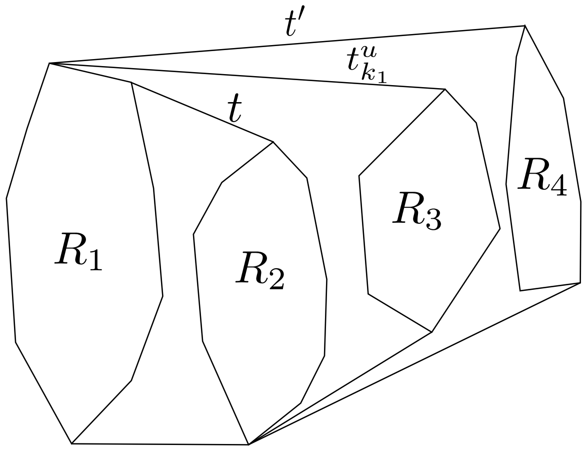


Figure 3.5: Example of the application to Lemma 3.10. In this case with only two iterations we find the tangent t' that covers t.

In Figure 3.5 we see an example of how the proof of how Lemma 3.10 works. In this case $t \notin \mathrm{UH}\,(Q_5)$ hence we find the iteration there t is involved in the merge phase, in this example in the third iteration, hence $k_1 = 3$. After the merge phase of the third iteration, the tangent $t_{k_1}^u$ covers t but $t_{k_1}^u \notin \mathrm{UH}\,(Q_5)$, hence we repeat the same process from $t_{k_1}^u$. In the fourth iteration, $k_2 = 4$, we find the tangent t' that covers $t_{k_1}^u$ and $t_{k_1}^u \in \mathrm{UH}\,(Q_5)$. As we have said in the proof, t' covers also t thus the proof ends.

Using Lemma 3.10 we can show a more restricted relationship that is valid only when the tangent t considered is covered by t_i^u. From Lemma 2.8 during the merge phase the only edges involved are the edges of $\mathrm{CH}(Q_i)$ visible from R_i. In particular if we focus only on the portion of $\mathrm{UH}(Q_i)$ visible from R_i, we have that every edge of this portion is covered by t_i^u. From the definition of M_i^u, we have that M_i^u contains every tangent that lies in $\mathrm{UH}(Q_i)$ that is involved in the merge phase. After this observation we have the following lemma.

Lemma 3.11. *For each phase i and tangent $t \in \left\{t_1^u, \cdots, t_{i-1}^u\right\}$ such that t_i^u covers t, there exists a tangent $t' \in M_i^u$ such that t' covers t*

Proof. From Lemma 3.10 exists a tangent $t' \in \left\{t_1^u, \cdots, t_{i-1}^u\right\}$ such that t' covers t and $t' \in \mathrm{UH}(Q_i)$:

$$\exists\, t' \in \left\{t_1^u, \cdots, t_{i-1}^u\right\} : \left(t' \text{ covers } t\right) \wedge \left(t' \in \mathrm{UH}(Q_i)\right)$$

If $t' \in M_i^u$ we have done. Otherwise if $t' \notin M_i^u$ this means that t' is not involved in the merge phase, hence if y_i is the extreme point of t_i^u in Q_i we have that t' is an upper hull edge of Q_i after y_i in the counterclockwise direction. Let be v is the vertical line that intersect y_i. Then the segment t' lies on the left of v whereas the segment t_i^u lies on the right. Moreover, since t' covers t then also t lie on the left of v, hence t_i^u do not covers t. But from the hypothesis t_i^u covers t, hence this is impossible. □

Another result related to the Lemma 3.10 regards the position of the tangents. Now we show another relationship between the tangents. In this case we show that exist a relationship between the positions where the tangents are placed, they cannot be located in every position in the plane.

Lemma 3.12. *Given two distinct tangents $t_1, t_2 \in T^u$, $t_1 \neq t_2$, then t_1 covers t_2 or t_2 covers t_1 or exist a vertical line l such that t_1 and t_2 lie on different half planes defined by l.*

Proof. Suppose that t_2 is added into the triangulation after t_1, in particular suppose $t_2 = t_i^u$ and $t_1 = t_j^u$, $i > j$. Consider Q_{i+1}, we have that t_2 lie on the upper hull of Q_{i+1} because the common upper tangent of the two sets Q_i and R_i in the previous merge phase. From Lemma 3.10 exists a tangent $t' \in \{t_1^u, \cdots, t_i^u\}$ such that t' covers t_1 and $t' \in \mathrm{UH}(Q_{i+1})$:

$$\exists\, t' \in \{t_1^u, \cdots, t_i^u\} : \left(t' \text{ covers } t\right) \wedge \left(t' \in \mathrm{UH}(Q_{i+1})\right)$$

The two segments t' and t_2 lies on the upper hull of Q_{i+1}, then we have two cases: $t' = t_2$ or $t' \neq t_2$. In the first case t_2 covers t_1. In the second case exist a vertical line l such that t' and t_2 lies on different half planes, but t_1 is covered by t' hence also these segments lies on different half planes of l.

If t_1 is added into the triangulation after t_2 we have that t_1 covers t_2 or exist a vertical line l such that t_1 and t_2 lie on different half planes defined by l. □

After have shown these relationships between the tangents we are ready to prove a similar relationship between the sets $N_1^u, \cdots, N_{n/s^u}^u$ as we have done for the sets $M_1^u, \cdots, M_{n/s}^u$.

Lemma 3.13. *Consider the sets $N_1^u, \cdots, N_{n/s}^u$, then:*

1. $N_i^u \cap N_j^u = \emptyset \qquad \forall\, i, j,\ i \neq j$

2. $\sum_{i=1}^{n/s} |N_i^u| \leq n$

Proof. Consider the sets N_i and N_j such that $i > j$. We consider $p \in N_i$ then from Lemma 3.12 we have two possible cases: exist a vertical line l such that t_i^u and t_i^u lie on different half planes defined by l or t_i^u covers t_j^u. In the first case we have $p \notin N_j$ by the definition. If t_i^u covers t_j^u then by Lemma 3.11 exists $t' \in \left\{t_1^u, \cdots, t_{i-1}^u\right\}$ such that t' covers t_j^u and $t' \in \mathrm{UH}\left(Q_i\right)$:

$$\exists\, t' \in \left\{t_1^u, \cdots, t_{i-1}^u\right\} : \left(t' \text{ covers } t_j^u\right) \wedge \left(t' \in \mathrm{UH}\left(Q_i\right)\right)$$

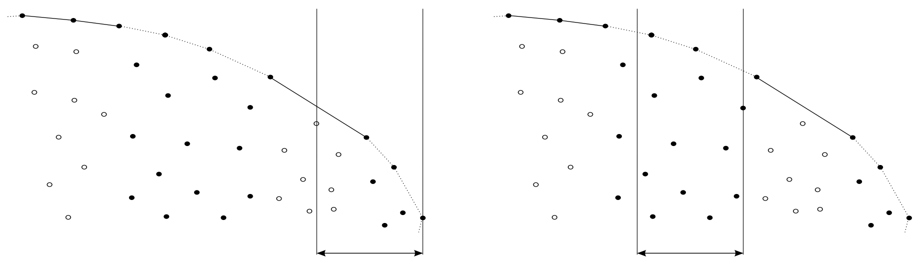
Therefore, $p \notin N_j$ by definition because t' covers every point of N_j,then we have $N_i^u \cap N_j^u = \emptyset$.

The second point derive directly from the first point:

$$\sum_{i=1}^{n/s} |N_i^u| = \left| \bigcup_{i=1}^{n/s} N_i^u \right| \leq n$$

□

Now we enough elements to show the complexity of Algorithm OPTIMAL_MULTI_PASS_TRIANGULATION. During the ith iteration of Algorithm OPTIMAL_MULTI_PASS_TRIANGULATION, Algorithm OPTIMAL_MERGE calls Algorithm PARTIAL_UPPER_HULL in order to scan the partial upper hull of Q_i visible from R_i. We remind that first we scan the portion of the upper hull of Q_i visible from R_i and then the portion of the lower hull of Q_i visible from R_i. We consider only the scan of the partial upper hull of Q_i. If t_i^u is the upper common tangent of Q_i and R_i, the upper partial hull to scan is the portion of $\mathrm{UH}\left(Q_i\right)$ that concerns the points of Q_i below t_i^u. We also remind that Algorithm PARTIAL_UPPER_HULL takes as input the set of the points P and a node of the upper hull v. The output of this algorithm is a sequence $\langle q_1, \cdots, q_r\rangle$ that is the portion of the upper hull from v in the counterclockwise direction (from the right to the left). Then when the merge phase related to the points in $\langle q_1, \cdots, q_r\rangle$ is done, Algorithm PARTIAL_UPPER_HULL is called again from q_1 in order to compute the next portion of the upper hull.



(A) The vertical slab contains at least one point of X_i^1.

(B) The vertical slab contains only points of X_i^2.

FIGURE 3.6: Two examples of the execution of Algorithm PARTIAL_-UPPER_HULL in order to scan the upper partial hull of Q_i during the ith merge phase. The two vertical line defines the vertical slab that contains the points of P returned by Algorithm LEFT_SLAB.

First we bound the number of calls of Algorithm PARTIAL_UPPER_HULL.

Lemma 3.14. *The overall number of calls of Algorithm* PARTIAL_UPPER_-HULL *done by Algorithm* OPTIMAL_MULTI_PASS_TRIANGULATION *is* $O(n/s)$.

Proof. Let be X_i the points of Q_i below t_i^u, these are the points that will be scanned by Algorithm PARTIAL_UPPER_HULL. Note that this algorithm could be scan also other points, but this will happen only at the last iteration hence it is at most one extra call. We partition X_i in two sets: X_i^1 and X_i^2, the first set contains the points below the tangents in M_i and the second set contains every point in N_i. This partition is valid from the definition of X_i, N_i and M_i^u. See Figure 3.4 to better understand this partition.

Let be $\langle q_1, \cdots, q_m \rangle$ the portion of the partial upper hull returned by a call of Algorithm PARTIAL_UPPER_HULL during the ith merge phase. At each call of this algorithm we have two cases depends on the points of Q_i that are below the portion of the upper hull $\langle q_1, \cdots, q_m \rangle$:

1. if there is at least one point of X^1 below $\langle q_1, \cdots, q_m \rangle$, then $\langle q_1, \cdots, q_m \rangle$ contains at least one edge of M_i^u;

2. if every points below $\langle q_1, \cdots, q_r \rangle$ is in X_i^2, then $\langle q_1, \cdots, q_r \rangle$ covers at least s points of N_i by Property 3.2.

Hence during the ith iteration, Algorithm PARTIAL_UPPER_HULL scan the partial upper hull of Q_i visible from R_i with at most $O(|N_i^u|/s + |M_i^u| + 1)$ iterations (we remind that there may be an extra iteration at the last step).

Hence during the ith iteration, Algorithm OPTIMAL_MULTI_PASS_TRIANGULATION calls Algorithm PARTIAL_UPPER_HULL in order to scan the partial upper hull of Q_i at most $O(|N_i^u|/s + |M_i^u| + 1)$ times.

Summing up all iteration and from Lemma 3.9 and 3.13, we have:

$$\sum_{i=0}^{n/s} O\left(\frac{|N_i^u|}{s} + |M_i^u| + 1\right) = O\left(\frac{n}{s}\right)$$

As we have said at the beginning of the analysis this results are valid also for the partial lower hull to scan at each iteration, hence the overall number of calls of PARTIAL_UPPER_HULL remains $O(n/s)$. □

Theorem 3.15. *Given a set P of n points in the plane and a parameter s, $1 \leq s \leq n$, Algorithm* OPTIMAL_MULTI_PASS_TRIANGULATION *outputs the triangulation of P in $O(n(n/s + \log s))$ time and $O(s)$ space with $O(n/s)$ passes of P.*

Proof. From Lemma 3.1 and 3.14 the overall cost of the merge phase is:

$$O\left(\frac{n}{s}(n + s\log s)\right)$$

The overall cost to find the next slab and compute the triangulation R_i during every phase is

$$\sum_{i=1}^{n/s} O(n + s\log s) = O\left(\frac{n}{s}(n + s\log s)\right)$$

Hence the total cost of Algorithm OPTIMAL_MULTI_PASS_TRIANGULATION is

$$O\left(\frac{n}{s}(n + s\log s)\right)$$

From Lemma 3.1, each call of Algorithm PARTIAL_UPPER_HULL takes $O(1)$ passes of P. Hence we have that the merge phases require a total of $O(n/s)$ passes of P. Moreover at each iteration of Algorithm OPTIMAL_-MULTI_PASS_TRIANGULATION we perform two addional scans of P: the first to aqcuire the next s points to compute (the sets $\{R_1 \cdots R_{n/s}\}$), and the second to update v. Since there are $O(n/s)$ iteration there are $O(n/s)$ addional passes. Then Algorithm OPTIMAL_MULTI_PASS_TRIANGULATION takes a total of $O(n/s)$ passes. □

Algorithm OPTIMAL_MULTI_PASS_TRIANGULATION has the same performance of the well known algorithm for sorting [21]. Moreover no algorithm with a substantially better time-space tradeoff is possible, due to known lower bounds for sorting [10]. Moreover, we remark that Algorithm OPTIMAL_MULTI_PASS_TRIANGULATION requires only $O(n/s)$ passes of P hence we have an upper bound to the number of reads of P.

Chapter 4

A parallel triangulation algorithm

In this chapter we shall present a parallel algorithm under our memory-constrained parallel computational model for triangulating a point set in the plane.

This algorithm derives directly from our optimal multi-pass algorithm that works on the memory constrained model. We keep the same structure as a multi-pass algorithm, hence at each pass we have to:

1. Find the next s points to compute, we call this point set R
2. Compute the triangulation of R
3. Merge the triangulation of the points in R with the triangulation of the points considered in the previous phases

To identify the points previously considered we store a variable v that contains the leftmost point in the set R. Our approach is to simply give a parallel algorithm for each of these steps.

4.1 Find the next points to compute

In order to find the next s points (if possible) to compute we use the same approach of the serial algorithm. We remind that in the serial algorithm (Algorithm RIGHT_SLAB), we have used a buffer of size $2s$ that contains the current s leftmost points in P at the right of v. We now show a parallel algorithm, named PARALLEL_RIGHT_SLAB, that is the parallel version of Algorithm RIGHT_SLAB.

Algorithm Imagine that the points in $P = \{p_1, \cdots, p_n\}$ are partitioned into n/s blocks $B_1, \cdots, B_{n/s}$ such that all blocks, except the last one, contains s points.

The algorithm scans the blocks $B_1, \cdots, B_{n/s}$, and for each block it performs the following operations:

1. store the points in the current block B_i into the buffer
2. sort the points in the buffer by x-coordinate
3. remove every point that lies on the left of v, including v itself if found
4. if the buffer contains al least s points, remove every point after the sth smallest point on the buffer

At the end of the algorithm, the buffer will contain the s leftmost points (if possible) in P that lie on the right of v.

We can implement the buffer using an array A of fixed length $2s$ and a variable k that stores the index of the first non empty cell of A. Therefore, the points currently stored in A are stored in $A[1] \cdots A[k-1]$. Step 1 can be performed by a parallel copy from P to the buffer. Step 2 is done with a classic sorting algorithm, for example the *pipelined merge sort* algorithm explained in [18]. Step 3 can be done in the following way: first we search the leftmost point that lies on the right of v, let z be the index of this point in A, second we remove every point stored in $A[1] \cdots A[z]$ by shifting the points $A[z+1] \cdots A[k-1]$ to the beginning of the buffer. Finally, step 4 can be easly done by updating k. The search operation can be done using the *parallel binary search* algorithm [18].

The pseudocode of this function is given by Algorithm 9. Function COPY (X, Y, p) copies every element from array Y to array X using p processors, SORT (X, p) is the function that sorts the array X with p processors, while BINARY_SEARCH (X, v, p) implements the parallel binary search of [18] using p processors. If v is not stored in X, BINARY_SEARCH returns the index of the rightmost point in X between the ones that lies on the left of v.

Lemma 4.1. *Given a set of n points P, a point $v \in P$ and two parameters p and s, $p \leq s$, we can output the s leftmost points (if possible) in P that lie on the right of v in $O((n \log s)/p)$ time and $O(s)$ space on the CREW PRAM model with p processors. The algorithm makes $O(n)$ operations on the read-only memory and $O(n \log s)$ operations on the shared memory.*

Proof. We first prove by induction on the number of iterations that after the generic ith iteration, the buffer will contain the s rightmost points in $B_1 \cup \cdots \cup B_i$ that lie on the right of v, if they exist. Otherwise, the buffer contains every point in $B_1 \cup \cdots \cup B_i$ that lies on the right of v.

Algorithm 9 Return the s leftmost points at the right of v in P

```
function PARALLEL_RIGHT_SLAB($P, v, s, p$)
    $A \leftarrow *$ array of size $2s$ in the shared memory $*$
    COPY ($A[1] \cdots A[s], P[1] \cdots P[s], p$)
    $k \leftarrow s + 1$
    for $i \leftarrow 1$ to $n/s - 1$ do
        COPY ($A[k] \cdots A[k + s - 1], P[si + 1] \cdots P[s(i + 1)], p$)
        SORT ($A[1] \cdots A[k], p$)
        $z \leftarrow$ BINARY_SEARCH ($A[1] \cdots A[k], v, p$)
        COPY ($A[1] \cdots A[k - z - 1], A[z + 1] \cdots A[k - 1], p$)
        $k \leftarrow k - z$
        if $k > s + 1$ then
            $k \leftarrow s + 1$
        end if
    end for
    return $A[1] \cdots A[k]$
end function
```

During the first iteration the buffer is empty, hence at the end of step 3 the buffer contains only points in B_1 that lie on the right of v. If the buffer contains less than s points we have finished, otherwise (step 4) the algorithm keeps only the s leftmost points and removes the others.

We suppose now that our statement is valid for every hth iteration, $h < i$, and let us consider the ith iteration. Initially the buffer contains the leftmost points that lie on the right of v in $B_1 \cup \cdots \cup B_{i-1}$. After step 3 the buffer contains the same points as before, plus every point of B_i that lies on the right of v. Therefore, the buffer contains the s leftmost points that lie on the right of v of $B_1 \cup \cdots \cup B_i$ plus some points of B_i which lies on the right of v. If the points in the buffer are at most s the ith iteration is finished, otherwise (step 4) the algorithm keeps only the s leftmost points in the buffer.

We now prove the complexity of this algorithm. Since all blocks are approximately of equal size, the size of each block B_i is $|B_i| = O\left(n/\left(n/s\right)\right) = O\left(s\right)$. Therefore, for each step of this algorithm we have the following complexities. The parallel copy can be done in $O\left(s/p\right)$ time and $O\left(p\right)$ space with p processors. The *pipelined merge sort algorithm* takes

$$O\left(\frac{s \log s}{p} + \log s\right) = O\left(\frac{s \log s}{p}\right)$$

time and $O\left(s\right)$ space. The parallel binary search takes

$$O\left(\frac{\log\left(s+1\right)}{\log\left(p+1\right)}\right)$$

time, and shifting the points it takes $O\left(\left(k-z\right)/p\right)=O\left(s/p\right)$ time, since $k-z\leq 2s$; the space required by these two last operations is indeed $O\left(p\right)$. Finally, step 4 clearly takes $O\left(1\right)$ time and space.

Therefore, each iteration takes

$$O\left(\frac{s}{p}+\frac{s\log s}{p}+\frac{\log\left(s+1\right)}{\log\left(p+1\right)}\right)$$

time and $O\left(s\right)$ space. But

$$\frac{\log\left(s+1\right)}{\log\left(p+1\right)}\leq\frac{\left(p+1\right)\log\left(s+1\right)}{p}\leq\frac{\left(s+1\right)\log\left(s+1\right)}{p}=O\left(\frac{s\log s}{p}\right)$$

hence we have

$$O\left(\frac{s}{p}+\frac{s\log s}{p}+\frac{\log\left(s+1\right)}{\log\left(p+1\right)}\right)=O\left(\frac{s\log s}{p}\right)$$

since $1\leq p\leq s$. The number of blocks if n/s, therefore the overall algorithm takes

$$O\left(\frac{n}{s}\left(\frac{s\log s}{p}\right)\right)=O\left(\frac{n\log s}{p}\right)$$

time. Each step requires at most $O\left(s\right)$ space, thus also the overall algorithm requires at most $O\left(s\right)$ space.

Regarding the data movement we have a total of $O\left(s\right)$ operations from the read-only memory and $O\left(s\log s\right)$ operations on the shared memory for each iteration of this algorithm. Since the number of iterations is n/s, we have a total of $O\left(n\right)$ operations on the read-only memory and $O\left(n\log s\right)$ operations on the shared memory. □

4.2 Parallel triangulation

The parallel triangulation can be done with an algorithm that works under the classical CREW PRAM model. For example we can use the algorithm of Ed Merks [20]. This algorithm computes the triangulation of an arbitrary point set of s points in $O\left(\log s\right)$ time using $O\left(s\right)$ space and $O\left(s\right)$ processors on the CREW PRAM model.

Unfortunately we cannot directly use this algorithm because we have only $p\leq s$ processors. However, we can easly adapt this algorithm in order to use less processors. In [18] is shown that every algorithm that runs in $T\left(s\right)$ time using $P\left(s\right)$ processors on the PRAM model can be adapted to use $p\leq P\left(s\right)$ processors and $O\left(C\left(s\right)/p+T\left(s\right)\right)$ time, where $C\left(s\right)=P\left(s\right)\cdot T\left(s\right)$ represents the cost of the algorithm.

In our case $T(s) = \log s$ and $C(s) = s \log s$, therefore we can compute the triangulation in

$$O\left(\frac{s \log s}{p} + \log s\right) = O\left(\frac{s \log s}{p}\right)$$

time using p processors and $O(s)$ space. Regarding the data movements, we have only $O(s \log s)$ operations on the shared memory because in this case the read-only memory is not involved.

Lastly we show how to find the next points to compute and how to connect the triangulating of these points with the ones previously computed.

4.3 Parallel merge

Also in this case we keep the same approach used in the serial algorithm. We refer to the second approach described, the one that uses Algorithm PARTIAL_UPPER_HULL to scan the partial hull of the points previously considered. Let us remind that Q_i is the point set that contains the points previously considered in the ith phase, while R_i is the point set of the points currently considered in the ith phase.

Now we shall first describe how to perform this function on our PRAM model, then we shall show how to report the triangles that connect the triangulations of the sets Q_i and R_i in the ith phase.

4.3.1 Parallel partial hull computation

We remind that, in the serial algorithm, Algorithm PARTIAL_UPPER_HULL works in this way:

1. compute the s rightmost points between the points that lie on the left of the point v, and let this set be Q
2. compute the upper hull of Q, and let w be the leftmost point in Q
3. adjust the upper hull of Q considering also the points that lie on the left of w

The parallel algorithm, let us call it PARALLEL_PARTIAL_UPPER_HULL, works in the same way except that every step is done in parallel. The first step can be done using the same approach of Algorithm PARALLEL_RIGHT_SLAB. The second step can be done using a classical upper hull algorithm that works on the CREW PRAM model. In [18] it is shown that finding the upper hull of a set of s points with p processors on the CREW PRAM model takes

$$O\left(\frac{s \log s}{p} + \log s\right) = O\left(\frac{s \log s}{p}\right)$$

time and $O(s)$ space.

Now before explaining how to do the third step, we need to make some observations about Algorithm PARTIAL_UPPER_HULL:

- it considers only the points that lie on the left of the left wall of Q, and it does not need to sort these points
- at the end of the algorithm, the only edge that does not lies on the original upper hull of Q is the last edge of the final sequence
- we can store the current status of the algorithm with only two variables that represent the last edge of the sequence; we do not need to modify the original sequence $\langle q_1, \cdots, q_r \rangle$ (see Figure 4.1).

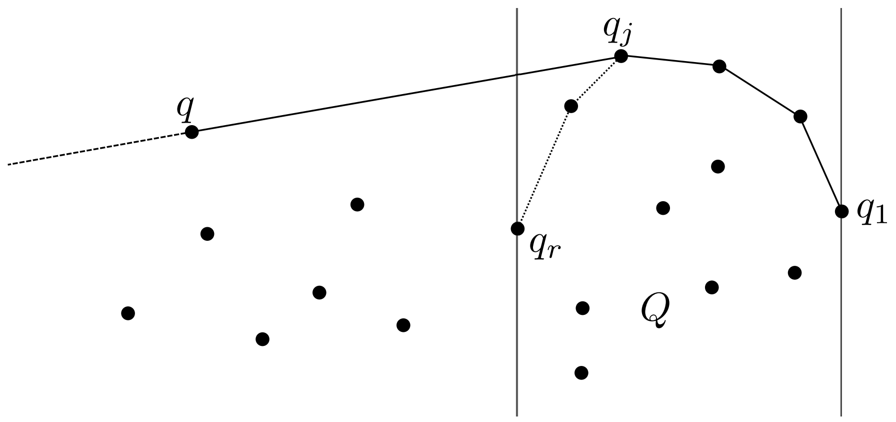


FIGURE 4.1: During Algorithm PARTIAL_UPPER_HULL, for storing the current status of the algorithm we only need to keep the last point q and the index j where the last segment of the sequence connects to $\langle q_1, \cdots, q_r \rangle$.

We now explain how to perform the third step of this algorithm.

Third step Due to these observations we can see Algorithm PARTIAL_-UPPER_HULL as a search algorithm, where the object to search is the segment $\overline{qq_j}$ (which is represented by the point q and the index j). This search operation can be carried out in the sequential model using two variables q and j that represent the current segment to return. This sequential algorithm performs a single scan of P, and for each point p that lies on the left of the left wall of Q it operates the following operations:

1. compute the segment $\overline{pq_k}$ that is tangent to the sequence $\langle q_1, \cdots, q_r \rangle$
2. if p, q and q_j makes a left turn then we set $q = p$ and $j = k$

This algorithm can be implemented using $O(s)$ space because we only need to store the sequence $\langle q_1, \cdots, q_r \rangle$ and the current tangent $\overline{qq_j}$ to return.

Our parallel algorithm is based on the same approach of this sequential algorithm. The idea is to use the balanced binary tree technique [18], which is a general strategy to design parallel algorithms for computing search operations. The main characteristic of algorithms based on this technique is that the DAGs associated to these algorithms are balanced binary trees. In our case we have p processors available, hence the DAG of our parallel algorithm will have p leaves and $p-1$ internal nodes. The input P is equally partitioned on the leaves, and each node of the tree represents a set of operations and holds information concerning the data stored at the leaves of the subtree rooted at u. The operations associated at the nodes in the DAG of the same level can be computed in parallel on different processors.

Let us explain now how our algorithm uses the balanced binary tree technique. For each node u, let R^u be the point set that contains all the points associated at the subtree rooted at u; this set represents the portion of P associated at the node u. The information associated at each node u consists in two variables q^u and r^u, that represent the last segment of the upper hull of $R^u \cup Q$ (the leftmost in our case).

- q^u stores the last end vertex of this segment (the leftmost vertex)
- r^u stores the index of the vertex in $\langle q_1, \cdots, q_r\rangle$ which is connected to q^u.

Therefore, the sequence $\langle q_1, \cdots, q_{r^u}, q^u\rangle$ represents the portion of the upper hull of $R^u \cup Q$ (from the right to the left) from the starting point v. The last segment is $\overline{q^u q_{r^u}}$. If u is a leaf of the tree, we directly compute the segment $\overline{q^u q_{r^u}}$. If u is an internal node of the tree, we compute the segment $\overline{q^u q_{r^u}}$ by performing the comparing operation between the segments associated at the left and at the right children of u.

We explain now how to implement this technique. Let $C_1, \cdots, C_p$ be the processors available in our model and let $l = n/p$, l is an integer number since n is a mutiple of p. We refer to the nodes of the tree with the pair (h, k), where h is the height of the tree and k is the index of the processor that compute the operations associated at (h, k). Moreover, let $R_{h,k}$ be the set that contains the points associated at (h, k). If $(h-1, 2k-1)$ and $(h-1, 2k)$ are the left and right children on (h, k), $R_{h,k}$ has the following definition:

$$\begin{cases} R_{h,k} = R_{h-1,2k-1} \cup R_{h-1,2k} & \forall\ 1 \leq h \leq \log p,\ 1 \leq k \leq p/2^h \\ R_{0,k} = \{p_{l(k-1)-1}, \cdots, p_{lk}\} & \forall\ 1 \leq k \leq p \end{cases}$$

Note that $R_{h,0} \cdots R_{h,p/2^h}$ form a parition of P for each level h of the tree, $0 \leq h \leq \log p$. Firstly, each processor C_k computes the operations associated at the leaves $(0,1) \cdots (0,p)$, hence it finds the segment $\overline{q^k q_{r^k}}$ based on the sequence $\langle q_1, \cdots, q_r\rangle$ and the point set $R_{0,k}$. After each processor C_k has finished, we traverse the tree from level 1 to level $\log p$ while computing the operations associated at the nodes. At each level h of the tree processors $C_1 \cdots C_{p/2^h}$ compute the operations associated at the nodes $(h,1) \cdots (h, p/2^h)$; note that processors $C_{p/2^h+1}, \cdots, C_p$ are not used. Each processor active C_k updates the variables q^k and r^k according to the segments stored in the left and right children of (h,k), these segments are stored in the variables q^{2k-1}, r^{2k-1}, q^{2k}, r^{2k}. In Figure 4.2 we can see an example when $p = 8$.

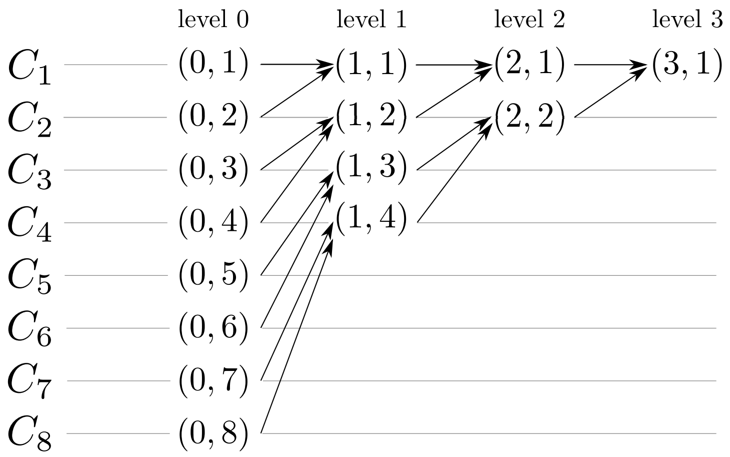


FIGURE 4.2: The balanced binary tree when $p = 8$.

Now we explain how to initialize the segments at the leaves and how to perform the comparing operation.

Initialize the segments at the leaves Given a leaf u of the tree, we computes q^u and r^u by scanning the set R^u. Initially we put $q^u = q_r$ and $r^u = s - 1$, with these values the segment $\overline{q^u q_{r^u}}$ is the last segment of the sequence $\langle q_1, \cdots, q_r \rangle$. For each point $p \in R^u$ that lies on the left of w, we perform a binary search into the sequence $\langle q_1, \cdots, q_r \rangle$ in order to find the point q_j, $2 \leq j \leq n-1$ such that:

- the points q^u, q_j and q_{j-1} make a nonleft turn
- the points q^u, q_{j+1} and q_j make a left turn.

If this point is found we select q_j. If the points q^u, q_j and q_{j-1} make a nonleft turn for every j, $2 \leq j \leq r$, then we select q_r. On the other hand, if the points q^u, q_{j+1} and q_j make a left turn for every j, $1 \leq j \leq r-1$, then we select q_1. Let q' be the point selected by the binary search.

After this operation we update the variables q^u and r^u only if q' comes before the point q_{r^u} in the sequence $\langle q_1, \cdots, q_r \rangle$. For updating q^u and r^u we only need to put $q^u = p$ and r^u to the index of q' in the sequence $\langle q_1, \cdots, q_r \rangle$.

Note that q^u and r^u are updated according to the same rule of the last step of Algorithm PARTIAL_UPPER_HULL, which is the step where we adjust the upper hull of Q considering also the points that lie on the left of w. The only difference is that in this case we perform a binary search for each point considered instead of linearly scannig the points (by making the pop operations on the stack).

Comparing operation Let u be an internal node, and let u_1,u_2 be the left and the right children of u. Also in this case we select q^u and r^u using the same approach of Algorithm PARTIAL_UPPER_HULL. Consider the portion of the upper hull that regards the set $R^{u_1} \cup Q$:

$$\langle q_1, \cdots, q_{r^{u_1}}, q^{u_1} \rangle$$

Depending on the type of turn that the points q^{u_2}, q^{u_1} and $q_{r^{u_1}}$ make, we choose the values of q^u and r^u according to the following cases:

1. q^{u_2}, q^{u_1} and $q_{r^{u_1}}$ make a left turn, $r^u = r^{u_2}$ and $q^u = q^{u_2}$
2. q^{u_2}, q^{u_1} and $q_{r^{u_1}}$ make a nonleft turn, $r^u = r^{u_1}$ and $q^u = q^{u_1}$

The pseudocode of the whole algorithm is given by Algorithm 10. We imagine that each processor executes the same algorithm and that the iterations of the for-loop are synchronized. In the PRAM model, all processors execute the same program such that, during each time unit, all the active processors are executing the same instructions, but on different data.

Lemma 4.2. *Given a set P of n points, a point v of the upper hull of P, and two parameters p and s, $1 \leq p \leq s$, then Algorithm* PARALLEL_PARTIAL_-UPPER_HULL *outputs the portion of the upper hull of P from the right to the left from v in $O((n \log s)/p)$ time and $O(s)$ space using p processors on the CREW PRAM model. The algorithm makes $O(n)$ operations on the read-only memory and $O(n \log s)$ operations on the shared memory.*

Proof. For demonstrating the correctness of this algorithm we only need to prove that, for each node u of the tree, the sequence $\langle q_1, \cdots, q_{r^u}, q^u \rangle$ represents the portion of the upper hull of $R^u \cup Q$ (from the right to the left) from the starting point v. We shall prove this statement by induction on the levels of the tree, where the first level of the tree consists of the leaves of the tree, and the last level is the root of the tree.

If u is a leaf (first level of the tree), after the init operation the sequence $\langle q_1, \cdots, q_{r^u}, q^u \rangle$ consists of the points of the upper hull of $Q \cup B = Q \cup R^u$. Let p be the current point considered and q' the point selected by the binary search. If q' comes before the current value q_{r^u} in the sequence $\langle q_1, \cdots, q_r \rangle$, we have that the points p, q_{r^u}, q_{r^u-1} make a left turn whereas the points q^u, q_{r^u}, q_{r^u-1} make a nonleft turn. This means that q^u and every point of

Algorithm 10 Compute a portion of the upper hull from v

```
function PARALLEL_PARTIAL_UPPER_HULL($P, v, p, s$)
    $*$ $j$ is the index of the processor $*$
    $Q \leftarrow$ PARALLEL_LEFT_SLAB($P, v, p, s$)
    $\langle q_1, \cdots, q_r \rangle \leftarrow$ PARALLEL_UPPER_HULL($Q, p$)
                                        ▷ Initialize the leaves at each processor
    $q^j \leftarrow q_r$
    $r^j \leftarrow r - 1$
    for $i \leftarrow 1$ to $n/p$ do
        $*$ update $q^j$ and $r^j$ according to the point $p_{\frac{n}{p}(j-1)+h}$ $*$
    end for
                                        ▷ Build the root of the bilanced binary tree
    for $i \leftarrow 1$ to $\log p$ do
        if $j \leq (p/2^h)$ then
            $(q^j, r^j) \leftarrow$ COMPARING_OPERATION $\big((q^{2j-1}, r^{2j-1}), (q^{2j}, r^{2j})\big)$
        end if
    end for
end function
```

the block B previously considered lie below the straight line that crosses p and q', hence the variables are updated (this situation is the same of Algorithm PARTIAL_UPPER_HULL when during the for-loop we make some pop operations on the stack). If q' comes after the value q_{r^u} in the sequence $\langle q_1, \cdots, q_r \rangle$, then q' lies below the straight line that crosses q^u and q_{r^u}, hence the variable it not updated.

Let us suppose now that our statement is true for every node contained in every h level, $h < i$. Let us consider now a node u in the ith level (u is an internal node). By induction, our statement is valid for the left and right children of u, u_1 and u_2, hence we have that:

- the sequence $\langle q_1, \cdots, q_{r^{u_1}}, q^{u_1} \rangle$ consists of the vertices of the upper hull of $R^{u_1} \cup Q$ (from the right to the left) from the starting point v
- the sequence $\langle q_1, \cdots, q_{r^{u_2}}, q^{u_2} \rangle$ consists of the vertices of the upper hull of $R^{u_2} \cup Q$ (from the right to the left) from the starting point v

If the comparing operation selects the segment $\overline{q^{u_1} q_{r^{u_1}}}$, the points q^{u_2}, q^{u_1} and $q_{r^{u_1}}$ make a right turn, hence the point q^{u_2} lies below the straight line that crosses q_{u^1} and $q_{r^{u_1}}$.

Moreover, from the induction hypothesis the segment $\overline{q^{u_2} q_{r^{u_2}}}$ is part of the upper hull of $R^{u_2} \cup Q$, hence every point $p \in R^{u_2}$ lies below the straight line that crosses the points q^{u_2} and $q_{r^{u_2}}$. Therefore, we have that every point $p \in R^{u_2}$ lies also below the straight line that crosses q^{u_1} and $q_{r^{u_1}}$ (this is true only because every point in $R^{u_1} \cup R^{u_2}$ lies on the left of the left wall of the point set Q). From the induction hypothesis, we have that every point $p \in R^{u_1} \cup Q$ lies below the straight line that crosses the points q^{u_1} and $q_{r^{u_1}}$ because the segment $\overline{q^{u_1} q_{r^{u_1}}}$ is part of the upper hull of $R^{u_1} \cup Q$. For what we have said we have that every point in $p \in R^{u_1} \cup R^{u_2} \cup Q$ lies below the straight line that crosses the points q^{u_1} and $q_{r^{u_1}}$. Therefore, we have that the sequence $\langle q_1, \cdots, q_{r^u}, q^u \rangle$ is the upper hull of the point set $R^{u_1} \cup R^{u_2} \cup Q = R^u$. The proof works on the same way of the other case of the comparing operation.

We shall study now the time and space complexities of this algorithm. From Lemma 4.1 and [18], finding the set Q and computing his upper hull takes

$$O\left(\frac{n \log s}{p} + \frac{s \log s}{p} + \log s\right) = O\left(\frac{n \log s}{p}\right)$$

time and $O(s)$ space using a classic PRAM algorithm for computing the upper hull [18]. Regarding the step 3 of the algorithm, we first have to initialize the segments of the leaves and then select the segment of the root of the tree by performing the comparing operations. The segments of the leaves can be computed in parallel with p processors since we have at most p leaves. All blocks are approximately of equal size, thus the size of each block B_i is $|B_i| = O(n/p)$. Therefore, the init operation takes

$$O\left(\frac{n \log r}{p}\right) = O\left(\frac{n \log s}{p}\right)$$

time and $O(s)$, since $r \leq s$ is the size of the upper hull of Q. For computing the segment at the root of the tree we only need to compute by level the segments at each node of the tree; from the second level (the first contains only the leaves) to the last level (the one that contains only the root). Each level consists of at most p nodes, hence each level can be computed in parallel in $O(1)$ time, since the comparing operation takes constant time. The number of levels is clearly $O(\log p)$, therfore we can select the segment of the root in

$$O(\log p)$$

time. The space required to adjust the upper hull of Q is the space required to store the sequence $\langle q_1, \cdots, q_r \rangle$ plus the space required to store the segments associated at the tree. The sequence takes at most $O(s)$ cells, whereas the tree has at most $O(p)$ nodes. In this way, the segments can be stored in $O(p)$ space, hence the step 3 takes $O(s)$. Therefore, we have that the overall algorithm takes

$$O\left(\frac{n \log s}{p} + \log p\right) = O\left(\frac{n \log s}{p}\right)$$

time and $O(s)$ space, since $1 \leq p \leq s \leq n$.

Regarding the data movement, from Lemma 4.1 and [18] the first two steps take $O(n)$ operations on the read-only memory and $O(s \log s)$ operations on the shared memory. Step 3 requires $O(n)$ operations on the read-only memory and $O(n \log s)$ operations on the shared memory, whereas the overall cost of the compare operations (for computing the segment at the root) requires no data movements because we can store the segments on the local memory of all processors. Therefore, Algorithm PARALLEL_PARTIAL_UPPER_HULL takes $O(n)$ operations on the read-only memory and $O(s \log s)$ operations on the shared memory. □

4.3.2 Compute the triangles of the merge phase in parallel

To compute the triangles of the merge phase we proceed on a different manner respect to the serial algorithm, but the main idea remains the same. Let Q_i be the point set that contains the points previously considered and R_i the point set that contains the points currently considered during the ith phase.

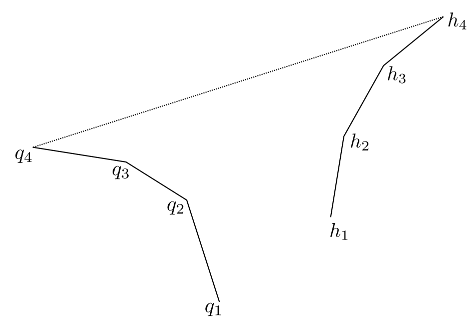
The idea remains the same used in the serial algorithm: first to compute and keep stored the convex hull of R_i, and then report the triangles that connect the two convex hulls of R_i and of Q_i while scannig the convex hull of Q_i. As we have done in the serial algorithm, we need to first consider the upper hulls of R_i and of Q_i, and then the lower hulls. These two operations remain the same, hence we explain only the merge phase that regards the upper hulls. The main difference between the parallel and serial merge algorithm is that in the parallel approach we report the triangles in blocks using another triangulation algorithm.

Let $\langle h_1, \cdots, h_l \rangle$ be the sequence of vertices of the upper hull of R_i from the left to the right, and since $|R_i| = O(s)$ we can explicitly store this sequence. Let $\langle q_1, \cdots, q_r \rangle$ be the sequence of vertices of the upper hull of Q_i from the left to the right. The algorithm is designed in order that $r \leq 2s \leq O(s)$, hence also this sequence can be explicitly stored. During the algorithm we remove some vertices from the beginning of these sequences, or we append some vertices at the end of these sequences.

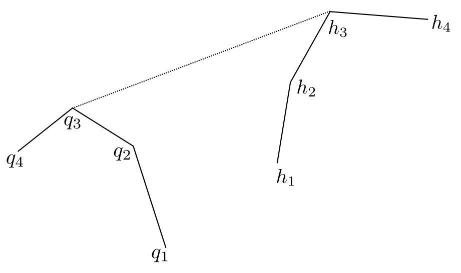
Algorithm Firstly we initialize the sequence $\langle h_1, \cdots, h_l \rangle$ with the vertices of the upper hull of R_i, whereas the sequence $\langle q_1, \cdots, q_r \rangle$ initially contains only the rightmost point of Q_i. Then, we proceed in this way:

1. find the next portion of the upper hull of Q_i using Algorithm PARALLEL_PARTIAL_UPPER_HULL from the vertex q_r, and append the sequence returned by the algorithm at the end of $\langle q_1, \cdots, q_r \rangle$

2. if $r \geq s$ then continue with the next steps, otherwise repeat from step 1



(A) In this case $r' = r$, hence the segment $\overline{q_{r'}h_{l'}}$ is not the upper common tangent of Q_i and R_i.



(B) In this $r' < r$, hence the segment $\overline{q_{r'}h_{l'}}$ is the upper tangent of Q_i and R_i.

FIGURE 4.3: Two possible cases of step 3. The dashed line represents the segment $\overline{q_{r'}h_{l'}}$.

3. compute the upper common tangent between $\langle h_1, \cdots, h_l\rangle$ and $\langle q_1, \cdots, q_r\rangle$ and let l' and r' be the indices of the end points of this tangent in the sequences

4. if $r' < r$ stop the process (the segment $\overline{q_{r'}h_{l'}}$ is the upper common tangent of Q_i and R_i, see Figure 4.3), otherwise continue to the next step;

5. compute the polygon triangulation of the polygon

$$V = \langle g_{r'}, \cdots, g_1, h_1, \cdots, h_{l'}\rangle$$

6. remove from $\langle g_1, \cdots, g_r\rangle$ the points $g_1, \cdots, g_{r'-1}$, and remove from $\langle h_1, \cdots, h_l\rangle$ the points $h_1, \cdots, h_{l'-1}$, and repeat from step 1

Note that for every step 3 we have $r' = r$ except the last one, were we have $r' < r$ and the algorithm stops to execute due to step 4. Moreover, step 6 is never executed during the last iteration, hence at the end of this step the sequence $\langle g_1, \cdots, g_r\rangle$ contains one point since $r' = r$.

Now we shall explain how to compute the upper common tangent between the two sequences and how to triangulate the polygon V.

Compute the upper common tangent In [18] is shown that we can compute the upper common tangent $\overline{q_{r'}h_{l'}}$ of the two sequences $\langle q_1, \cdots, q_r\rangle$ and $\langle h_1, \cdots, h_l\rangle$ in $O(1)$ time, with a linear work. The main idea is to perform a parallel search on $\langle q_1, \cdots, q_r\rangle$ in order to find $q_{r'}$ and then repeat the same process on $\langle h_1, \cdots, h_l\rangle$ in order to find $h_{j'}$. For using this algorithm in our parallel model we have to adapt it using the same method adopted before in order to use p processors. In this case the size of the polygon V is

$|P| = l + r \leq 2s = O(s)$, hence the work of this algorithm is $C(s) = O(s)$. Therefore, we can compute the upper common tangent between the two sequences $\langle q_1, \cdots, q_r \rangle$ and $\langle h_1, \cdots, h_l \rangle$ in

$$O\left(\frac{s}{p} + 1\right) = O\left(\frac{s}{p}\right)$$

time and $O(s)$ space in the CREW PRAM model with p processors. Moreover, we find out that each call of this algorithm takes only $O(s)$ memory operation on the shared memory. Let us call this algorithm PARALLEL_-UPPER_TANGENT.

Compute the polygon triangulation Before explaining how to compute the triangulation of $V = \langle g_{r'}, \cdots, g_1, h_1, \cdots, h_{l'} \rangle$, we firstly want to remark that the points of the polygon are sorted by the x-coordinate in increasing order:

$$x(g_{r'}) \leq \cdots \leq x(g_1) \leq x(h_1) \leq \cdots \leq x(h_{l'})$$

Therefore the polygon V is a monotone polygon respect to the x-axis. Since V is a monotone polygon, we can apply Algorithm CHAIN_TRIANGULATE of Atallah and Goodrich [5]. This algorithm correctly triangulates the polygon V under his side (which in our case is the x-axis) in $O(\log s)$ time and $O(s)$ space using $O(s)$ processors.

Also in this case we don't have enough processors available. Therefore, we have to adapt this algorithm in order to use p processors, applying the same method used before. The work of this algorithm is $C(s) = O(s \log s)$, hence Algorithm CHAIN_TRIANGULATE computes the triangulation of V in

$$O\left(\frac{s \log s}{p} + \log s\right) = O\left(\frac{s \log s}{p}\right)$$

time and $O(s)$ space with p processors on the CREW PRAM model. Regarding the memory operations, we have $O(s \log s)$ operations on the shared memory for each call of this algorithm. In the same way we report the triangles that connect the lower hulls of Q_i and R_i.

The pseudocode of the whole algorithm is given by Algorithm 11. We have included also the part that regards the lower hulls. The functions PARALLEL_UPPER_HULL and PARALLEL_LOWER_HULL return respectively the upper hull and the lower hull from the left to the right.

4.4 Total complexity

We shall give now the theorem of this chapter that shows the performance of our parallel algorithm.

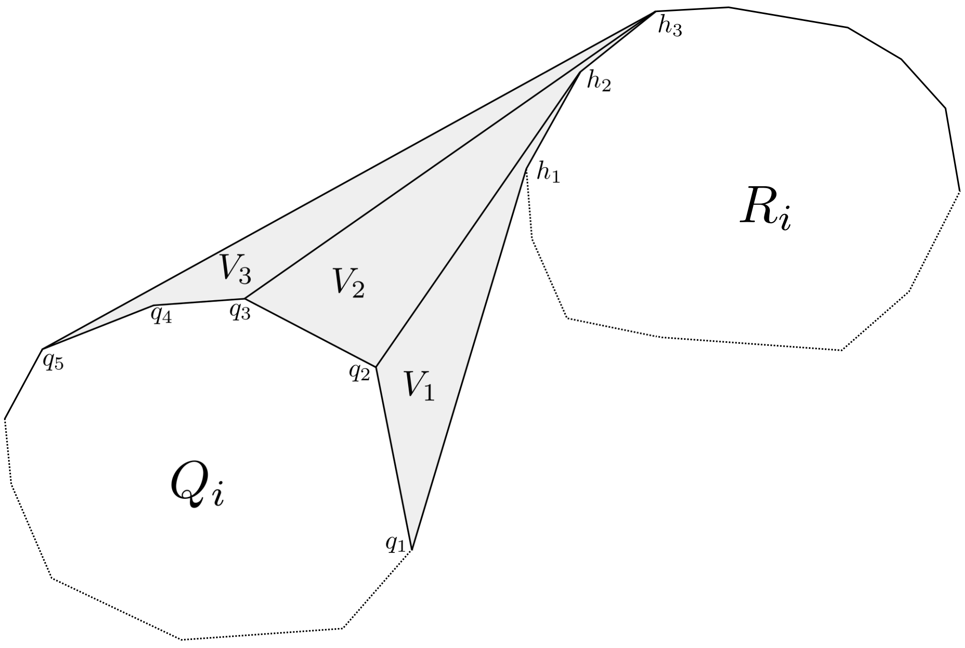


FIGURE 4.4: During this merge phase, the polygons V_1, V_2 and V_3 are found when we connects the upper hulls of Q_i and R_i.

Theorem 4.3. *Given a set P of n points and two integer values p and s, $1 \leq p \leq s$, we can report the triangulation of P in $O\left(\left(n^2 \log s\right) / (ps)\right)$ time and $O(s)$ space with p processors on the CREW PRAM model. The algorithm makes $O\left(n^2/s\right)$ operations on the read-only memory and $O\left(\left(n^2 \log s\right)/s\right)$ operations on the shared memory.*

Proof. Since Lemma 4.1 and 4.2 are valid, the proof of correctness remains the same because the algorithm works in the same way in comparison with the serial algorithm.

We shall now prove the complexity of this algorithm. For each phase i, to find the set R_i and to compute the triangulation of R_i it takes

$$O\left(\frac{n \log s}{p} + \frac{s \log s}{p} + \log s\right) = O\left(\frac{n \log s}{p}\right)$$

time. Since $p \leq s$ and the number of phases is $O(n/s)$, the overall time required among all phases is

$$O\left(\frac{n^2 \log s}{p \cdot s}\right)$$

Regarding the merge phase, we have the cost due to:

- scan the convex hull of Q_i with Algorithm PARALLEL_PARTIAL_UPPER_HULL

- compute the upper common tangent between $\langle q_1, \cdots, q_r \rangle$ and $\langle h_1, \cdots, h_l \rangle$ with Algorithm PARALLEL_UPPER_TANGENT
- compute the polygon triangulation of the polygon V with Algorithm CHAIN_TRIANGULATE

From Lemma 3.14 the overall number of calls to Algorithm PARALLEL_-PARTIAL_UPPER_HULL is $O(n/s)$. Therefore, the overall time taken by the merge phase due to the calls of this algorithm is

$$O\left(\frac{n^2 \log s}{p \cdot s}\right)$$

For what concerns the computation of the upper common tangent and the polygon triangulation, at each iteration of the merge algorithm we have that:

- we always have $r' = r \geq s$ except in the last iteration, hence $\langle q_1, \cdots, q_r \rangle$ and V has at least s points (the size of V is $|V| = r' + l' \geq r'$) except in the last iteration
- these algorithms are called always on different points

Since there are at most n points, these two algorithms are called at most $O(n/s)$ times, hence the overall time taken by the merge phase due to the calls of these algorithms is

$$O\left(\frac{n}{s}\left(\frac{s \log s}{p} + \frac{s}{p} + \log s\right)\right) = O\left(\frac{n}{s}\left(\frac{s \log s}{p}\right)\right) = O\left(\frac{n \log s}{p}\right)$$

Therefore, the overall cost of the merge phases among all phases is

$$O\left(\frac{n^2 \log s}{p \cdot s}\right)$$

time. Hence the overall parallel algorithm takes

$$O\left(\frac{n^2 \log s}{p \cdot s}\right)$$

time.

For what concerns the space required by the algorithm, let us first show that the size of $\langle q_1, \cdots, q_r \rangle$ is $r = O(s)$. The size of this sequence can increase only on step 1 due to Algorithm PARALLEL_PARTIAL_UPPER_HULL. From Lemma 3.1 this algorithm returns at most s vertices, hence the size of $\langle q_1, \cdots, q_r \rangle$ increases at most s. Step 1 can be executed in three different cases:

1. at the beginning of the algorithm, in this case we have $r = 1$

2. after step 2 when $r \leq s-1$

3. after step 6, in this case $r = 1$ because have removed every point from the sequence except the last (note that step 6 is executed only if we are not on the last iteration due to step 4)

Therefore, after appending the vertices returned by Algorithm PARALLEL_PARTIAL_UPPER_HULL into $\langle q_1, \cdots, q_r \rangle$, the size of this sequence can be at most

$$r + s \leq s - 1 + s \leq 2s - 1 = O(s)$$

Everyone of the just mentioned parallel algorithms takes $O(s)$ space and p processors on the CREW PRAM model. Moreover, no one of these algorithms is executed in parallel, therefore also the overall algorithm takes $O(s)$ space and p processors on the CREW PRAM model.

Regarding the memory operations, by Lemma 4.1 there are at most $O(n)$ operations on the read-only memory and $O(n \log s)$ operations on the shared memory due to find the set R_i and compute the triangulation of R_i. Therefore, the total amount of memory operations due to these operations is $O\left(n^2/s\right)$ for the read-only memory and $O\left(\left(n^2 \log s\right)/s\right)$ for the shared memory, since we have $O(n/s)$ phases. Moreover, we have seen before that Algorithms PARALLEL_PARTIAL_UPPER_HULL, CHAIN_TRIANGULATE and PARALLEL_UPPER_TANGENT are executed at most $O(n/s)$ times, hence by Lemma 4.2 the total amout of memory operations is $O\left(n^2/s\right)$ for the read-only memory and $O\left(\left(n^2 \log s\right)/s\right)$ for the shared memory. □

Algorithm 11 Report every triangle that connects Q_i to R_i

```
function PARALLEL_MERGE($P, v, R, p, s$)
    $\langle h_1, \cdots, h_l \rangle \leftarrow$ PARALLEL_UPPER_HULL($R, p$)
    $\langle g_1, \cdots, g_r \rangle \leftarrow \langle v \rangle$
    repeat
        $\langle u_1, \cdots, u_m \rangle \leftarrow$ PARALLEL_PARTIAL_HULL $(P, q_r, p, s)$
        $\langle g_1, \cdots, g_r \rangle \leftarrow \langle g_1, \cdots, g_r, u_1, \cdots, u_m \rangle$
        if $r \geq s$ then
            $(r', l') \leftarrow$ PARALLEL_UPPER_TANGENT $(\langle g_1, \cdots, g_r \rangle, \langle h_1, \cdots, h_l \rangle, p)$
            if $r' = r$ then
                $V \leftarrow \langle g_{r'}, \cdots, g_1, h_1, \cdots, h_{l'} \rangle$
                CHAIN_TRIANGULATE $(V, p)$
                $\langle g_1, \cdots, g_r \rangle \leftarrow \langle g_{r'}, \cdots, g_r \rangle$
                $\langle h_1, \cdots, h_l \rangle \leftarrow \langle h_{l'}, \cdots, h_l \rangle$
            else
                break
            end if
        end if
    until TRUE
    $\langle h_1, \cdots, h_l \rangle \leftarrow$ PARALLEL_LOWER_HULL($R, p$)
    $\langle g_1, \cdots, g_r \rangle \leftarrow \langle v \rangle$
    repeat
        $\langle u_1, \cdots, u_m \rangle \leftarrow$ PARALLEL_PARTIAL_HULL $(P, q_r, p, s)$
        $\langle g_1, \cdots, g_r \rangle \leftarrow \langle g_1, \cdots, g_r, u_1, \cdots, u_m \rangle$
        if $r \geq s$ then
            $(r', l') \leftarrow$ PARALLEL_LOWER_TANGENT $(\langle g_1, \cdots, g_r \rangle, \langle h_1, \cdots, h_l \rangle, p)$
            if $r' = r$ then
                $V \leftarrow \langle g_{r'}, \cdots, g_1, h_1, \cdots, h_{l'} \rangle$
                CHAIN_TRIANGULATE $(V, p)$
                $\langle g_1, \cdots, g_r \rangle \leftarrow \langle g_{r'}, \cdots, g_r \rangle$
                $\langle h_1, \cdots, h_l \rangle \leftarrow \langle h_{l'}, \cdots, h_l \rangle$
            else
                break
            end if
        end if
    until TRUE
end function
```

Chapter 5

Conclusion

In this thesis we have studied the triangulation problem, a fundamental primitive in computational geometry. Firstly we have showed a general but inefficient approach, then we have explained how to modify this algorithm in order to obtain an optimal algorithm. Then, we have showed how to use this algorithm in order to obtain a parallel algorithm.

These algorithms are developed on our memory-computational models, and they require $O(s)$ work-space cells of storage, where s is a parameter given as input for both the algorithms. Moreover, the time taken by these algorithms decreases when the amount of avaiable space increases. In particular, we have seen that the result obtained with the parallel algorithm can be adapted in order to match other computational models.

As future work, it would be of great interest to investigate if it is possible to modify these algorithms in order to output a specific triangulation, such as the Delaunay triangulation [16]. This particular triangulation has the following properties:

1. It maximizes the minimum angle of all the angles of the triangles in the triangulation
2. No point lies inside the circumcicle of any triangle of the triangulation.

Moreover, Delaunay triangulation is directly connected to Voronoi diagram [6], which is another classic computational geometry problem. These geometric structures have many applications in other areas, where it may be interesting to have an algorithm that fits the space avaiable for the computation.

The main problem of using our approach is that we have to satisfy the properties of these geometric structures respect to the whole set of points given as input. For example we cannot use a convex hull algorithm in order to perform the merge phase.

Another interesting challenge would be to create a parallel algorithm for problems for which a sequential memory-constrained algorithm already exists. Recently, Asano ed Al. [3] derive a space-efficient algorithm for the shortest path problem that takes $O(s)$ space. It would be usefull to have a space efficient parallel algorithm that works on our parallel computational model.

Bibliography

[1] AM Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[3] Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Gnter Rote, and Andr Schulz. Memory-constrained algorithms for simple polygons. *Computational Geometry*, 46(8):959 – 969, 2013.

[4] Tetsuo Asano, Wolfgang Mulzer, Günter Rote, and Yajun Wang. Constant-work-space algorithms for geometric problems. *JoCG*, 2(1):46–68, 2011.

[5] M J Atallah and M T Goodrich. Efficient plane sweeping in parallel. In *Proceedings of the second annual symposium on Computational geometry*, SCG '86, pages 216–225, New York, NY, USA, 1986. ACM.

[6] Franz Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991.

[7] Luis Barba, Matias Korman, Stefan Langerman, Kunihiko Sadakane, and Rodrigo I. Silveira. Space-time trade-offs for stack-based algorithms. *CoRR*, abs/1208.3663, 2012.

[8] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. *SIGPLAN Not.*, 31(6):213–225, June 1996.

[9] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, August 1973.

[10] A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM Journal on Computing*, 11:287–297, 1982.

[11] Herve Brnnimann, John Iacono, Jyrki Katajainen, Pat Morin, Jason Morrison, and Godfried Toussaint. Space-efficient planar convex hull algorithms. In *Proc. Latin American Theoretical Informatics*, pages 494–507, 2002.

[12] Herv Brnnimann and Timothy M. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. In Martn Farach-Colton, editor, *LATIN 2004: Theoretical Informatics*, volume 2976 of *Lecture Notes in Computer Science*, pages 162–171. Springer Berlin Heidelberg, 2004.

[13] Timothy M. Chan. Comparison-based time-space lower bounds for selection. *ACM Trans. Algorithms*, 6(2):26:1–26:16, April 2010.

[14] Timothy M. Chan and Eric Y. Chen. Multi-pass geometric algorithms. In *Proceedings of the twenty-first annual symposium on Computational geometry*, SCG '05, pages 180–189, New York, NY, USA, 2005. ACM.

[15] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[16] B. Delaunay. Sur la sphère vide. *Izvestia Akademia Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7:793–800, 1934.

[17] Xiaoqiu Huang. A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, 18(3):223–239, 1989.

[18] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[19] D.T. Lee and B.J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[20] Ed Merks. An optimal parallel algorithm for triangulating a set of points in the plane. *Int. J. Parallel Program.*, 15(5):399–411, October 1986.

[21] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, pages 253–258, Washington, DC, USA, 1978. IEEE Computer Society.

[22] J.Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theoretical Computer Science*, 165(2):311 – 323, 1996.

[23] Franco P. Preparata and Michael I. Shamos. *Computational geometry: an introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1985.