is parallel planar point location problem in NC

Yes — **the planar point location problem is in NC**.

What is the problem?

Given:

- A planar subdivision (e.g., a planar graph embedded in the plane),
- A query point q,

Determine which face (region) of the subdivision contains q.

Complexity Result

The planar point location problem **can be solved in NC**, meaning:

- **Polylogarithmic time**: $O((\log n)^k)$
- Using **polynomially many processors**

So it lies in **Nick's Class (NC)**.

Why is it in NC?

The key reasons:

1. Planar graph preprocessing can be done in NC

2. Planar separator theorems allow recursive decomposition
3. Planar graph reachability is in NC (via results of Omer Reingold and others for undirected connectivity)
4. Parallel tree contraction and centroid decomposition are in NC

Using these tools:

- Build a hierarchical decomposition (separator-based)
- Do parallel binary search over regions
- Each level takes polylog time
- Depth of recursion is $O(\log n)$

Total parallel time remains polylogarithmic.

More Precise Classification

Planar point location is in:

- **NC²** (classical results)
- Improved bounds exist depending on the model (PRAM variant)

It is certainly in **P**, and also in **NC**.

Important Distinction

- **Single query after preprocessing** → in NC
- **Building the full point location data structure** → also parallelizable in NC
- **Batch queries** → easily parallelizable

Comparison with Factorization (since you were discussing NC earlier)

Problem	In P	In NC
Planar point location	✔️	✔️
Integer factorization	❓ (believed yes)	❌ (not known)

Planar point location is much “more parallel-friendly” than factorization.

If you’d like, I can also:

- Sketch an explicit NC algorithm
- Compare it to parallel convex hull
- Or discuss lower bounds / completeness results

define the problem of parallel planar point location

Definition: Parallel Planar Point Location Problem

The **parallel planar point location problem** is the parallel version of the classical planar point location problem.

1️⃣ Input

1. A **planar subdivision** S:

- A planar straight-line embedding of a graph
- Induces faces (regions) in the plane
- Total size n (vertices + edges)

2. A set of query points

$$Q = \{q_1, q_2, \ldots, q_m\}$$

2️⃣ Output

For each query point q_i, determine:

- The **face (region)** of subdivision S that contains q_i.

Formally, compute a function:

$$f : Q \to \mathrm{Faces}(S)$$

Parallel Version (Complexity-Theoretic Form)

The problem is in **parallel complexity theory**:

> Design an algorithm that solves the above task in **polylogarithmic time** using **polynomially many processors**.

In complexity notation:

- Time: $O((\log n)^k)$

- Processors: $n^{O(1)}$

  If achievable → the problem is in **NC**.

Decision Version (Formal Language View)

To phrase it as a decision problem:

Given:

- Planar subdivision S
- Query point q
- Face identifier F

  Question:

  Does $q \in F$?

This version is useful when analyzing membership in complexity classes like **NC**, **P**, etc.

Variants

🔹 1. Single-query version

- One query point
- Parallelism used inside the search

🔹 2. Batch-query version

- Many query points

- Queries processed simultaneously

🔹 3. Preprocessing model

Two-stage formulation:

1. Preprocess subdivision S
2. Answer queries in parallel

Geometric Interpretation

The task is equivalent to:

- Determining in which polygon (face) a point lies
- Performing a sequence of orientation / half-plane tests
- Navigating a planar graph hierarchy

Relation to Other Parallel Problems

It is closely related to:

- Parallel planar graph reachability
- Parallel tree contraction
- Parallel separator decomposition

All known to be in NC.

Compact Formal Definition

Parallel Planar Point Location
Input: A planar subdivision S of size n and query points Q.
Output: For each $q \in Q$, identify the face of S containing q.
Goal: Solve in polylogarithmic time using polynomial processors.

If you want, I can next:

- Formalize it in PRAM model notation
- Express it as a circuit family definition
- Or give an explicit NC algorithm schema

rasterize a hyperbolic arc on two dimensional plane

Rasterizing a Hyperbolic Arc in 2D

1️⃣ What is a hyperbolic arc?

Typically, a hyperbolic arc is a portion of a curve defined by a hyperbola such as:

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$$

or equivalently

$$y = \pm b\sqrt{\frac{x^2}{a^2} - 1}$$

You are given:

- Parameters a, b
- Interval $x \in [x_{\min}, x_{\max}]$
- A 2D pixel grid

Goal:

Determine which pixels approximate the hyperbolic arc.

2️⃣ Rasterization Approaches

There are three main practical methods.

Method 1: Parametric Sampling (Simple & Practical)

Parameterize the hyperbola using hyperbolic functions:

$$x(t) = a \cosh t$$

$$y(t) = b \sinh t$$

Then:

1. Choose parameter range $t \in [t_1, t_2]$
2. Sample t at small increments

3. Map continuous (x, y) to pixel coordinates:

$$(i, j) = (\lfloor x \rfloor, \lfloor y \rfloor)$$

4. Mark those pixels

Pros

- Very simple
- Stable
- Easy to implement

Cons

- May produce gaps if sampling too coarse
- Needs adaptive step size for uniform density

Method 2: Implicit Function Rasterization (More Accurate)

Define implicit function:

$$F(x, y) = \frac{x^2}{a^2} - \frac{y^2}{b^2} - 1$$

A pixel (i, j) is on the arc if:

$$|F(i, j)| < \epsilon$$

Procedure:

1. Iterate over bounding box
2. Evaluate $F(i, j)$

3. Select pixels close to zero level set

This is analogous to:

- Implicit circle rasterization
- Conic section rendering

Method 3: Midpoint / Incremental Algorithm (Efficient)

Similar to Bresenham's circle algorithm:

1. Use decision function:

$$d = F(x + \frac{1}{2}, y - 1)$$

2. Choose next pixel based on sign of d
3. Update using incremental arithmetic

This avoids expensive square roots.

This method:

- Uses only addition/subtraction
- Suitable for hardware rendering
- Runs in linear time in number of pixels

3️⃣ Practical Algorithm (Recommended)

If implementing in software (Python, graphics, simulation):

Step-by-step:

1. Compute parameter bounds:

$$t_1 = \operatorname{arcosh}(x_{\min}/a)$$

$$t_2 = \operatorname{arcosh}(x_{\max}/a)$$

2. Adaptive step size:

$$\Delta t \approx \frac{1}{\max(|dx/dt|, |dy/dt|)}$$

3. For $t = t_1$ to t_2:

- Compute $x(t), y(t)$
- Plot pixel

4️⃣ Handling Both Branches

Hyperbola has two branches:

$$(x, y) = (\pm a \cosh t, \pm b \sinh t)$$

Choose branch depending on arc you want.

5️⃣ Complexity

If arc spans L pixels:

- Time: $O(L)$
- Space: $O(1)$

Parallel version:

- Each processor handles a parameter subinterval
- Achievable in **NC** (polylog time, polynomial processors)

6️⃣ Geometric Interpretation

Rasterizing hyperbolic arc is equivalent to:

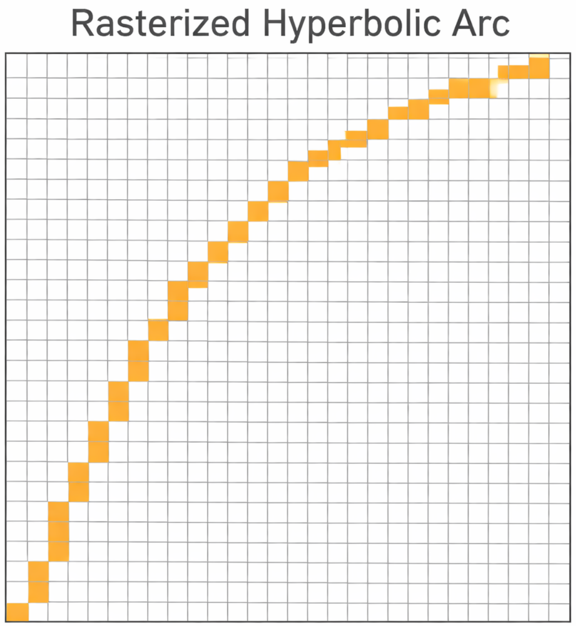
- Rendering a conic section
- Evaluating a quadratic form
- Sampling a level set of a polynomial

If you'd like, I can:

- Provide exact Python raster code
- Derive the midpoint incremental recurrence
- Or show how to do this using integer-only arithmetic (hardware style)

can you show a graphic example

Image created • **Rasterised hyperbolic arc on grid**


Rasterized Hyperbolic Arc