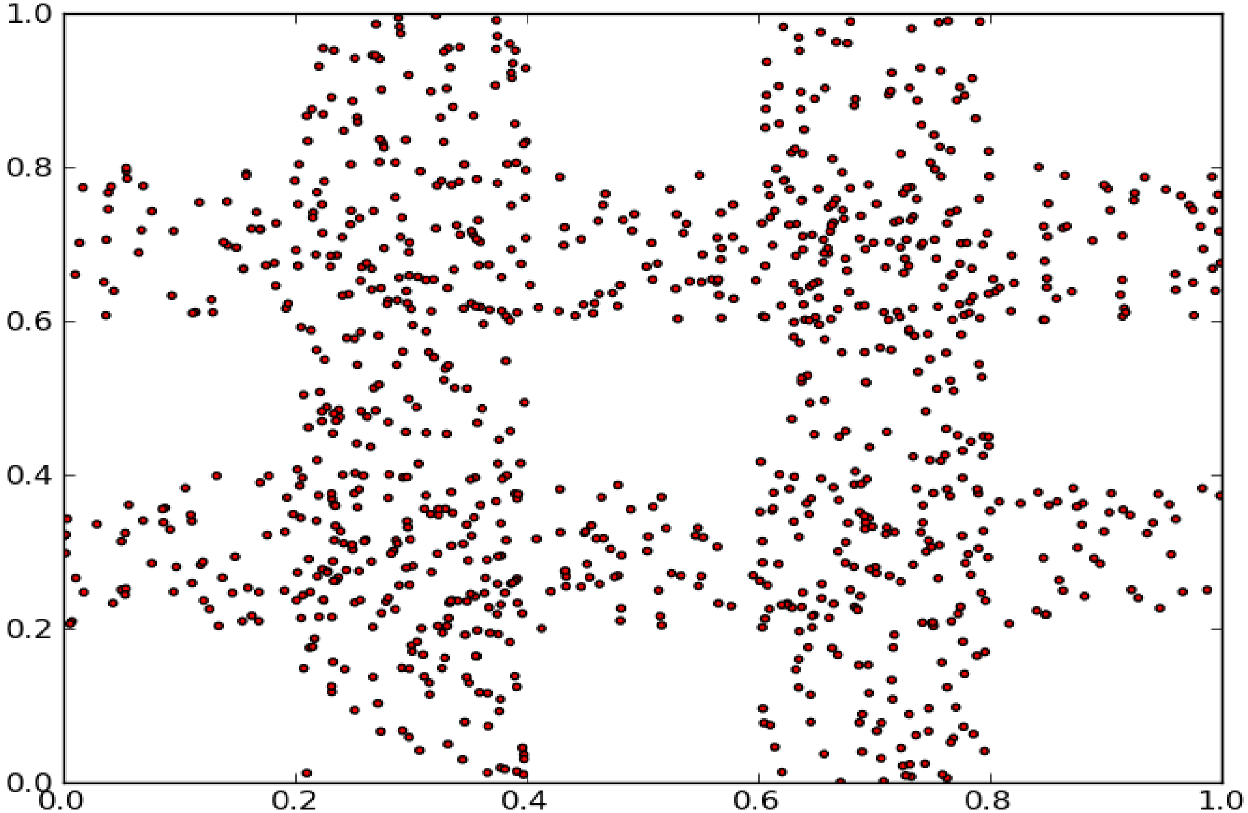
Parallelizing the Growing Self-Organizing Maps algorithm using Software Transactional Memory

Growing Self-Organizing Maps

Is a clustering algorithm.

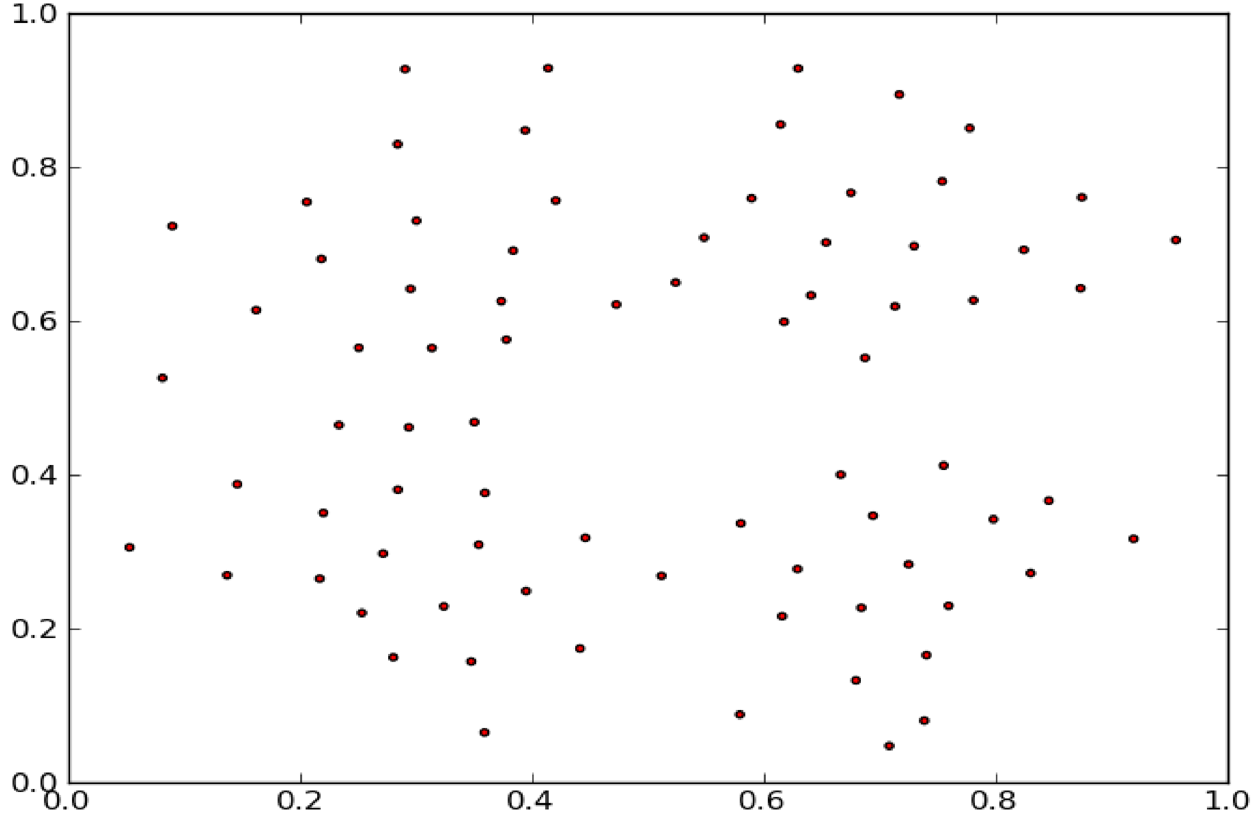
Growing Self-Organizing Maps

So for example this is what the input looks like:



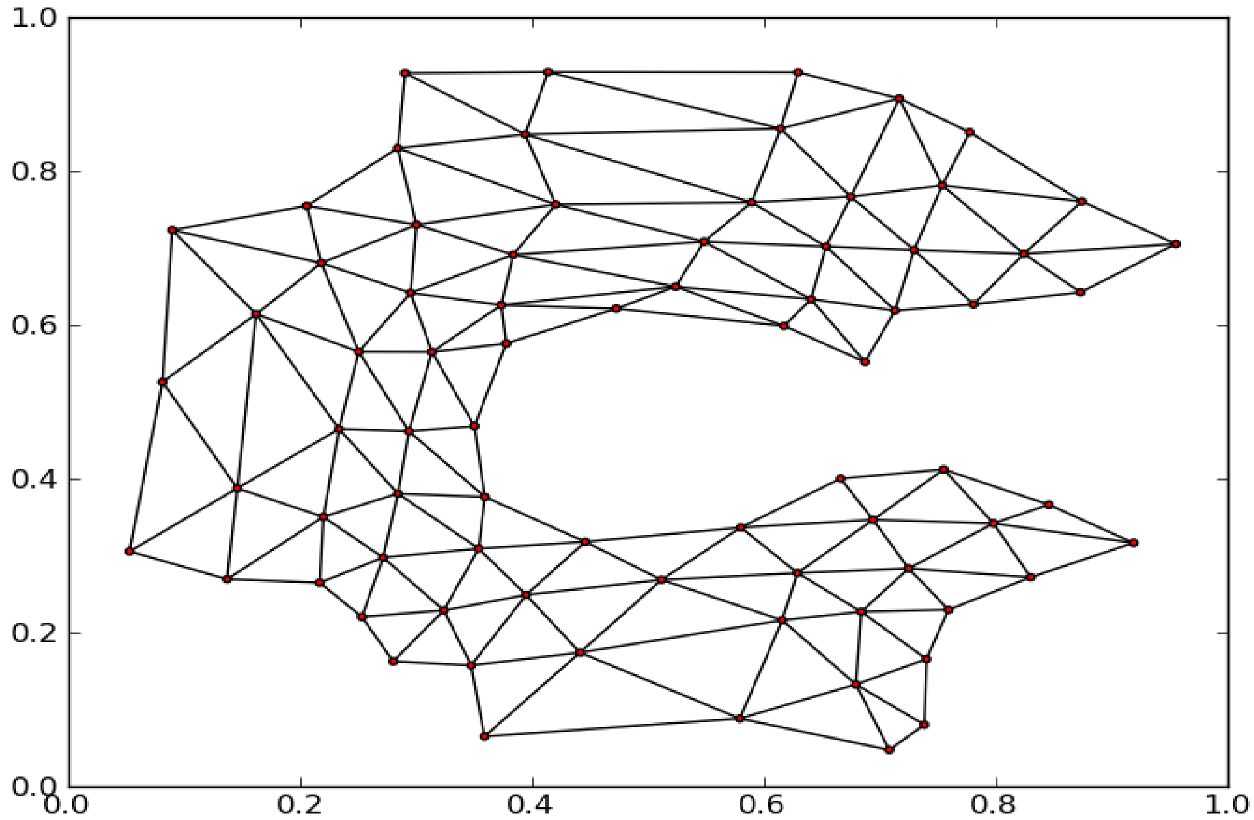
Growing Self-Organizing Maps

And this is the output you would get:



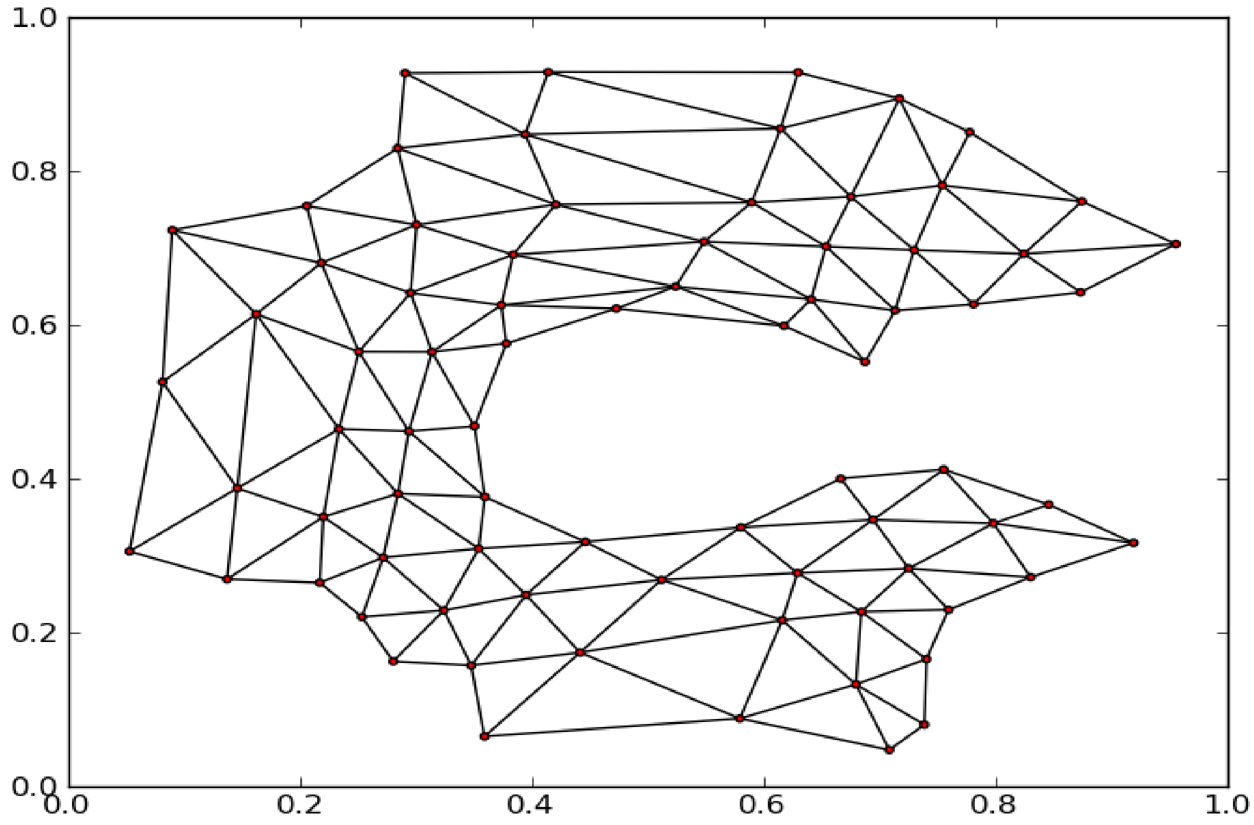
Growing Self-Organizing Maps

Bonus: output is a planar graph.



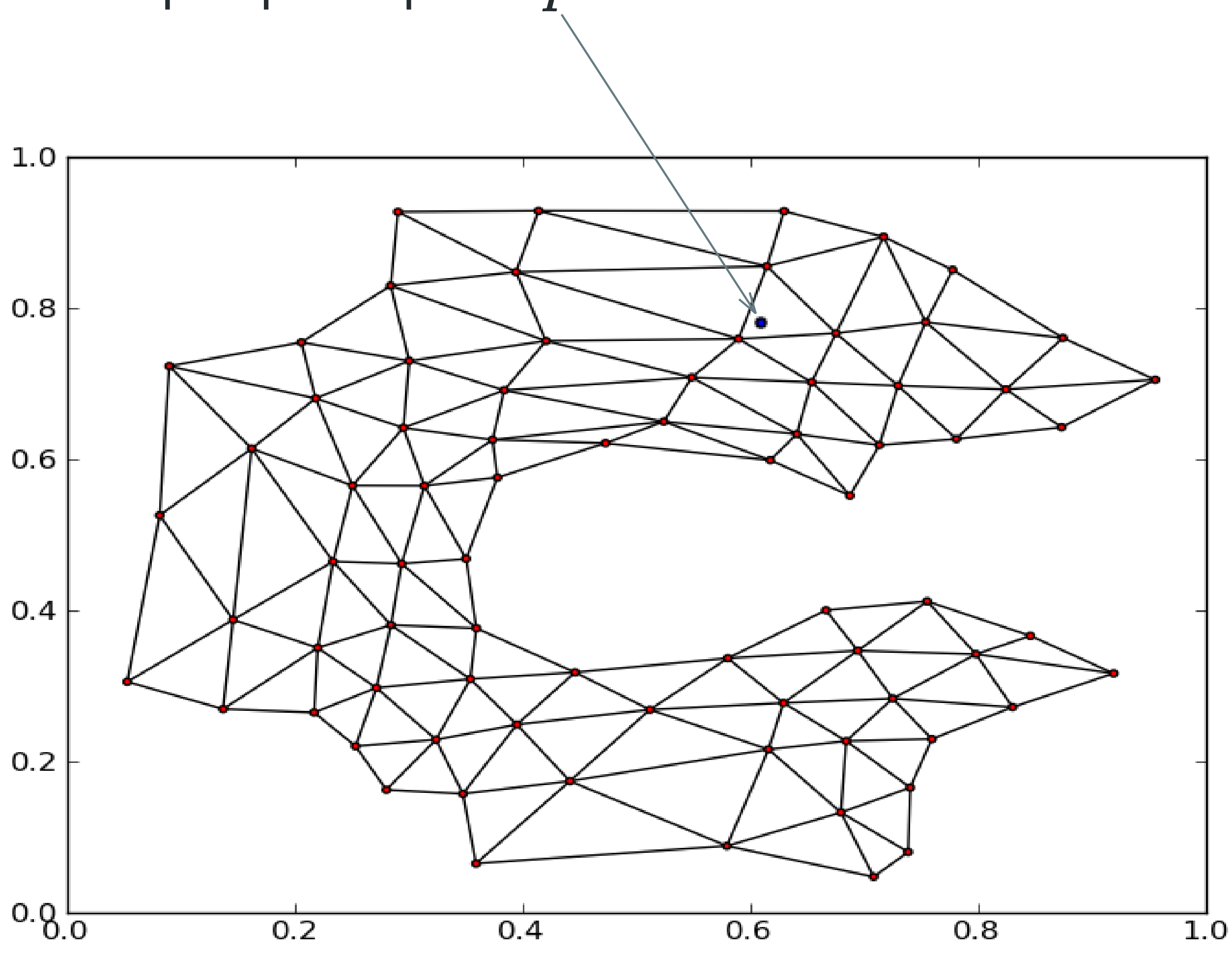
Growing Self-Organizing Maps

So how to you generate this output?



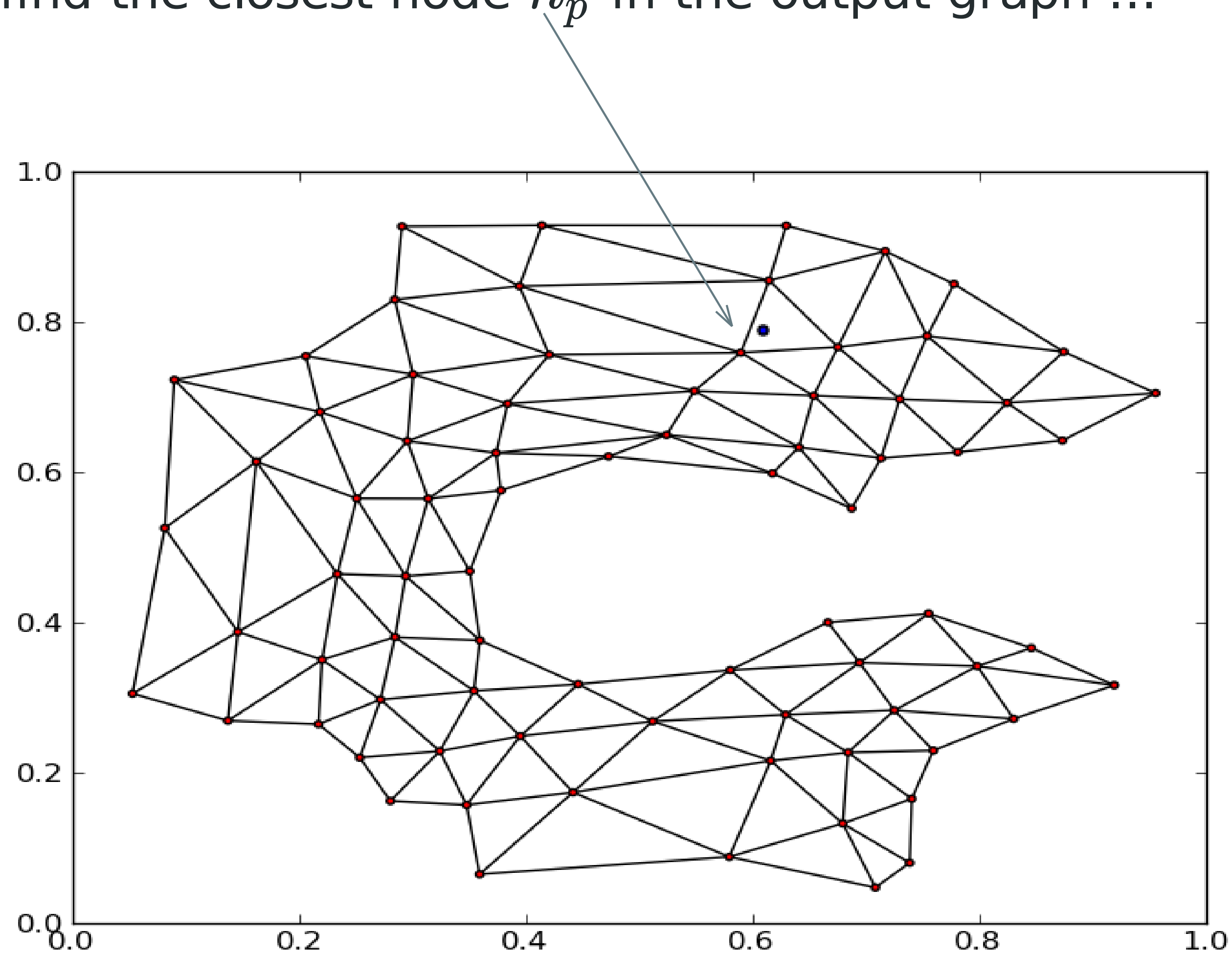
Growing Self-Organizing Maps

For each input point point p …



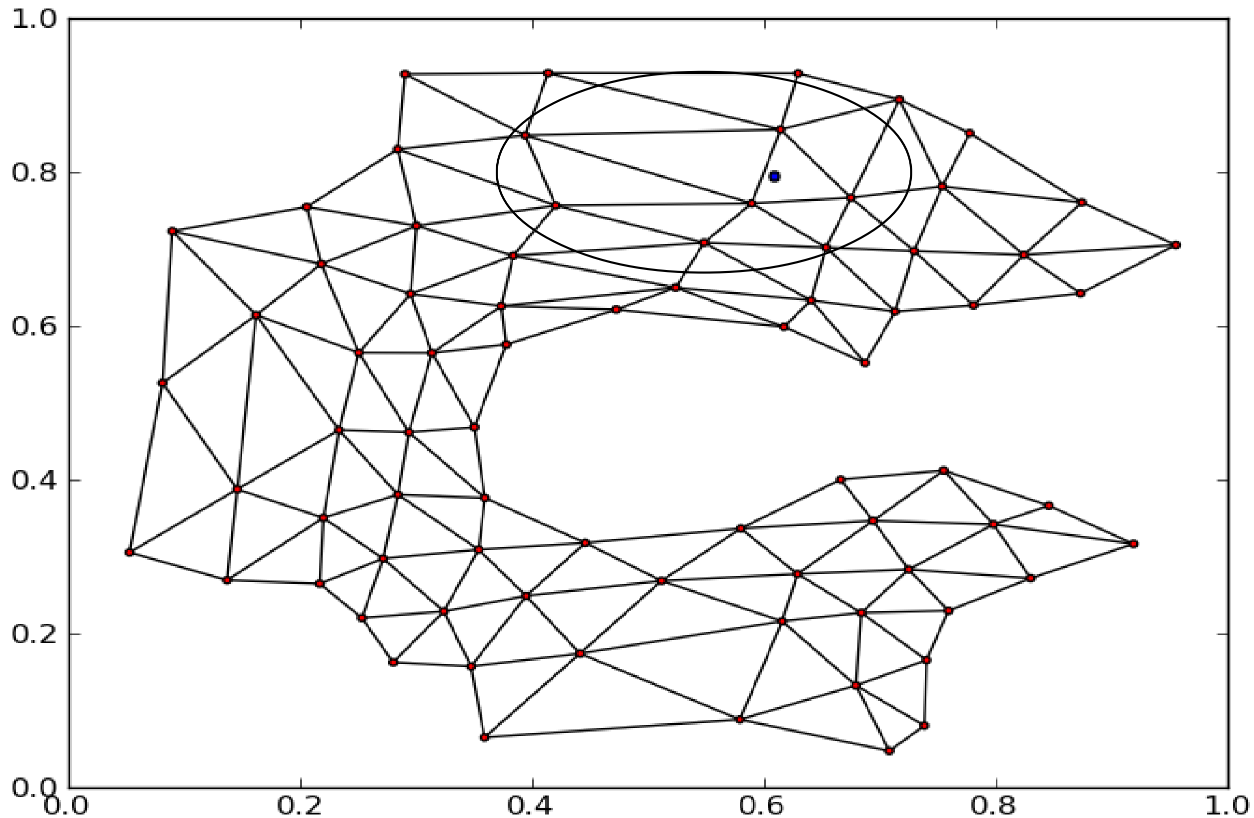
Growing Self-Organizing Maps

you find the closest node n_p in the output graph ...



Growing Self-Organizing Maps

and pull every node n' in a neighborhood of n_p closer to p.



Growing Self-Organizing Maps

Growth:

- start with a minimal number of nodes,
- keep track of the accumulated error for each node,
- check whether it exceeds a certain threshold,
- propagate the error to neighbours for internal nodes,
- create new neighbours for boundary nodes.

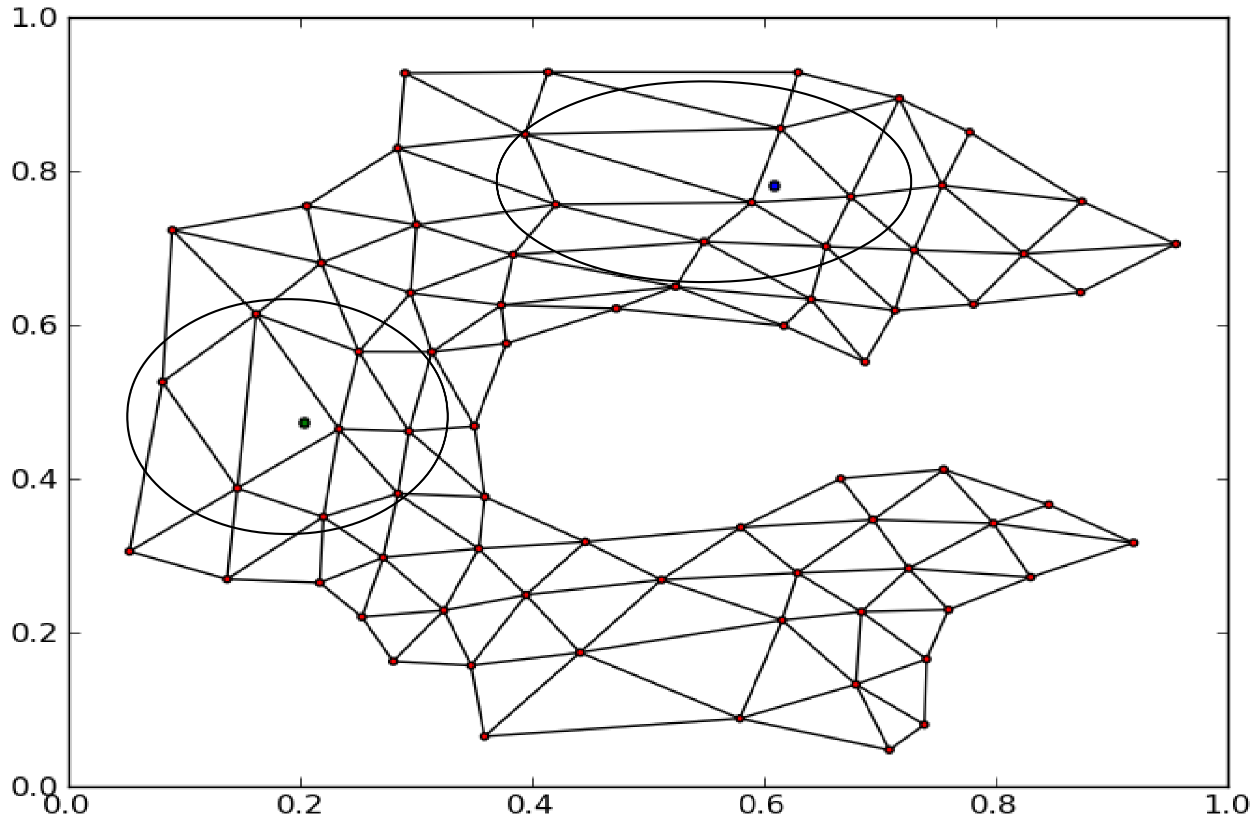
Growing Self-Organizing Maps

Parallelization:

- this thing is slow ($\sim O(n^2)$),
- need to exploit parallelization potential,
- special case considered here: Multiprocessor/Multicore systems,
- not GPUs,
- no distributed computing.

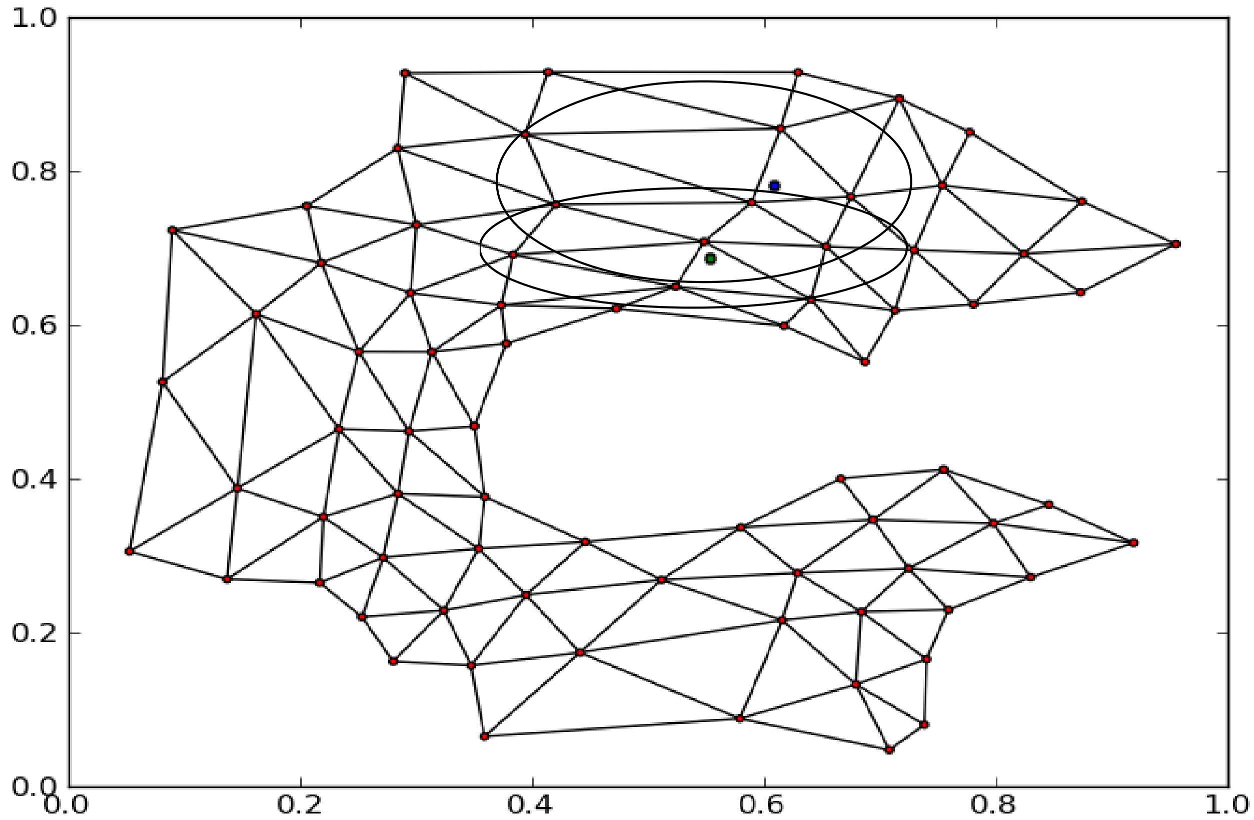
Growing Self-Organizing Maps

No problem:



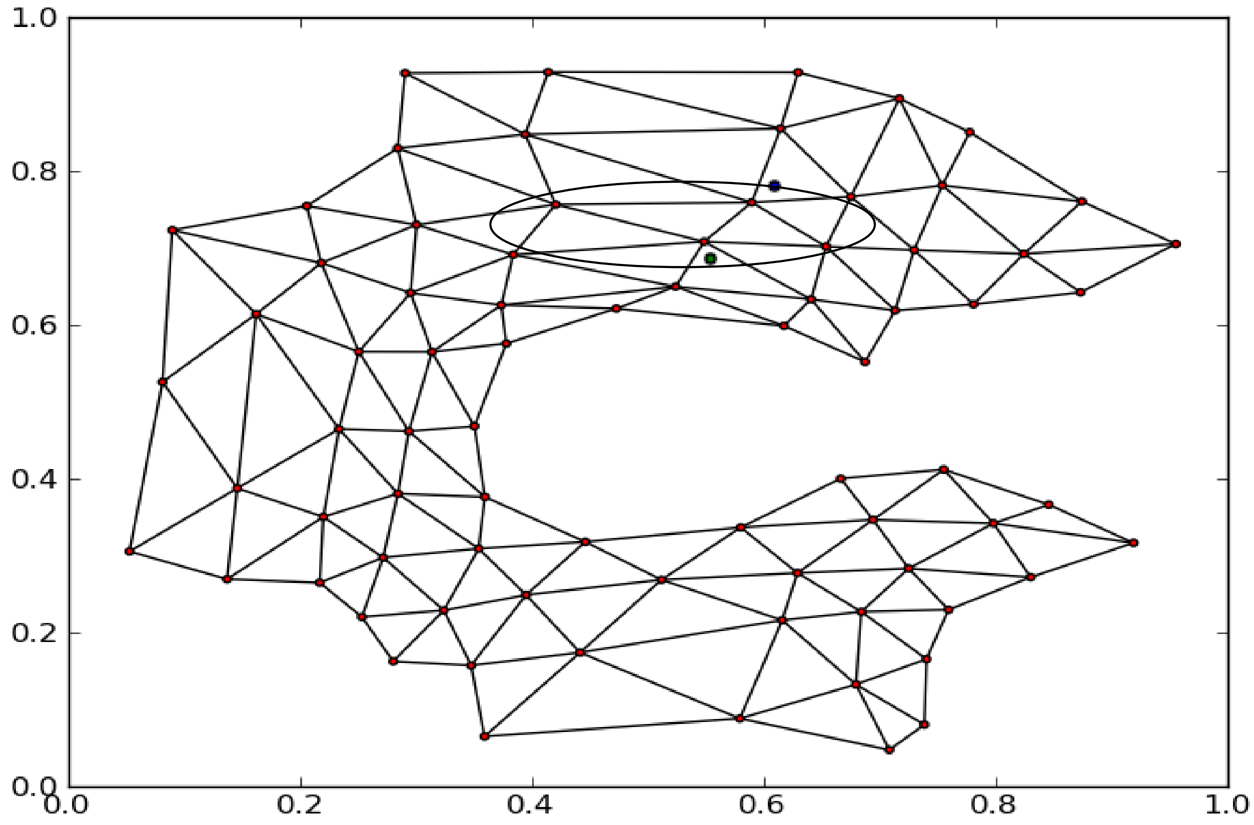
Growing Self-Organizing Maps

Problem:



Growing Self-Organizing Maps

Problem:



Growing Self-Organizing Maps

Problem:

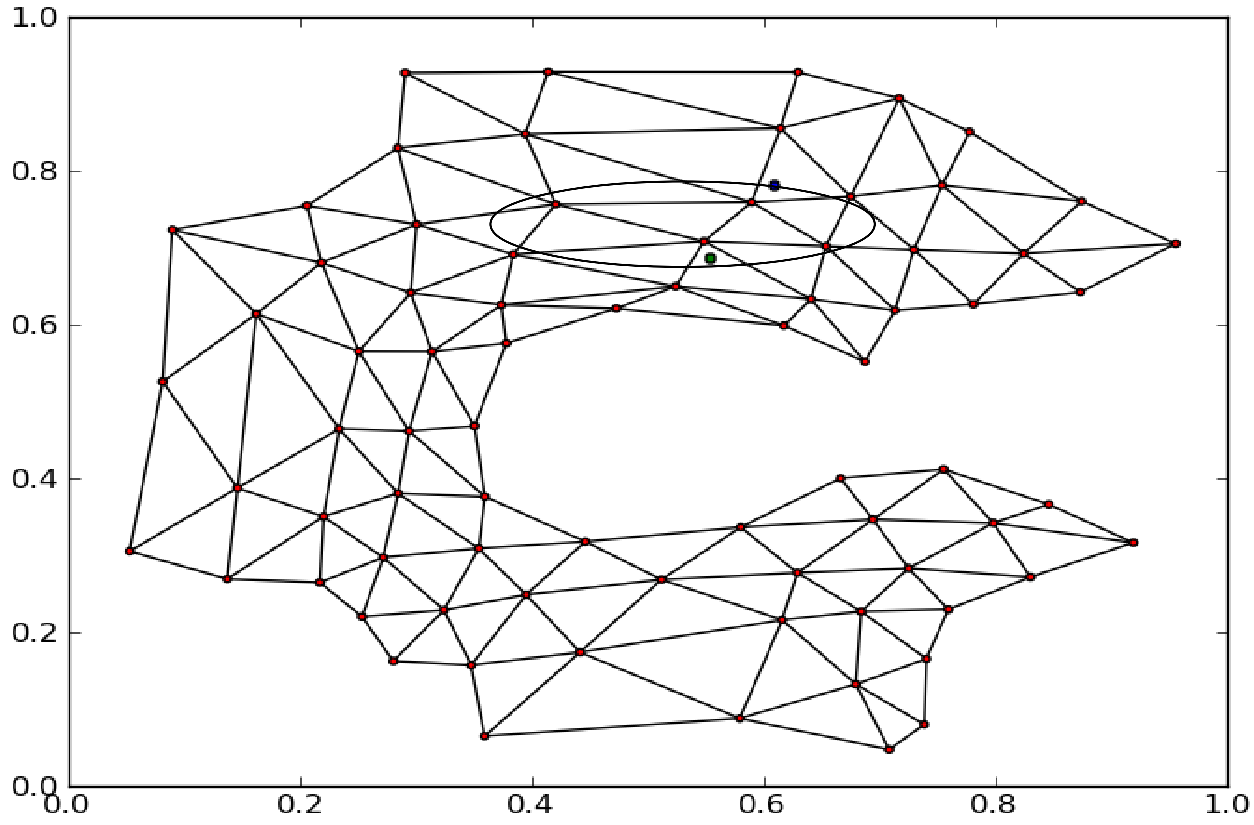
- need a way to synchronize parallel tasks.

Traditional solution:

- locks, semaphores, critical sections,
- get complex quickly,
- don't compose,
- error prone (deadlocks, livelocks, resource starvation, priority inversion)

Growing Self-Organizing Maps

Deadlock example (do you see the solution?):



Growing Self-Organizing Maps

Deadlock example (do you see the solution?)

Or: use a different concurrency abstraction, namely Software Transactional Memory.

Software Transactional Memory

is a concurrency abstraction that:

- brings transaction semantics known from databases to software/programming,
- was proposed in the 95s,
- can be implemented VERY differently,
- is easier to reason about than locking,
- keeps a shared memory model,
- doesn't use user level locks,
- is still an area of research.

Software Transactional Memory

Swapping the values of two variables:

```
swap a b = atomically (do
  value_a <- readTVar a
  value_b <- readTVar b
  writeTVar b value_a
  writeTVar a value_b)
```

Software Transactional Memory

also has limits:

- transactions mean restarts,
- restarts disallow side effects,
- restarts can have surprising performance characteristics.

Haskell's implementation:

- controls side effects through the type system,
- doesn't use locking,
- uses an optimistic approach.

Applying STM to GSOM

means figuring out:

- thread granularity,
- transaction granularity,
- invariants between transactions.

Applying STM to GSOM

Thread granularity:

- one point p per thread.

Transaction granularity:

- figure out n_p in one transaction (T_1),
- move n_p and its neighbors closer to p in another (T_2).

Applying STM to GSOM

Transaction invariant:

- n_p has minimum distance to p at the end of T_1 and at the beginning of $T2$,
- is ensured by keeping track of (p, n_p) pairs in a lookup table t,
- checking t whenever a node n is modified and updating t if necessary,
- modifications happen only during $T2$,
- transaction semantics guarantee correctness.

Results:

Around 20% speedup for 2 dimensions, 2 threads and 2 cores.
Why so slow?

- most expensive transaction is $T1$,
- $T1$ is highly likely to be restarted,
- restarts kill performance gains.

Results:

Even worse for higher dimensions (i.e. around 200):

- running time degenerates to being unusable.

But for this scenario a different parallelization strategy would be more appropriate:

- parallelize distance measure calculations (possibly on GPUs).

Thank you for your patience!