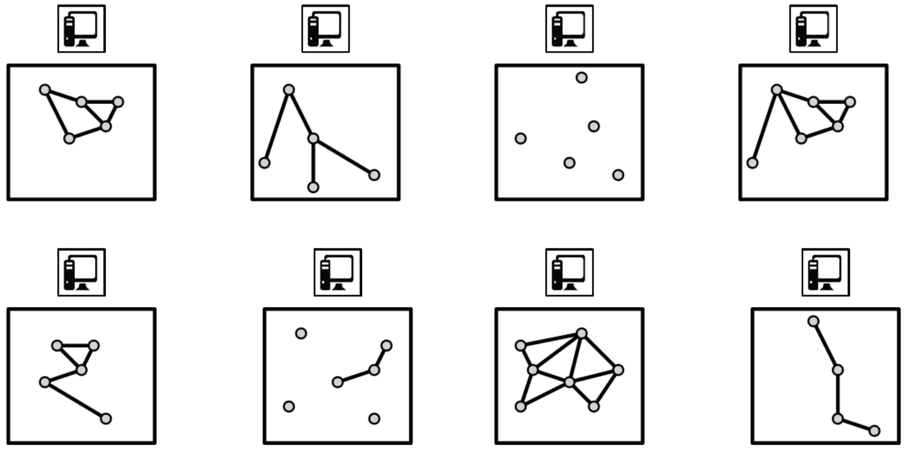
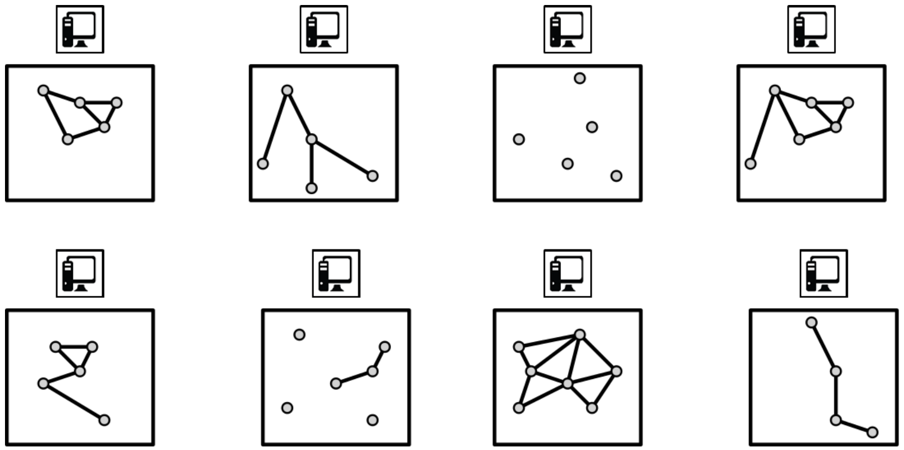
Parallel Algorithms for Local Problems in Sparse Graphs

Jara Uitto, Aalto University

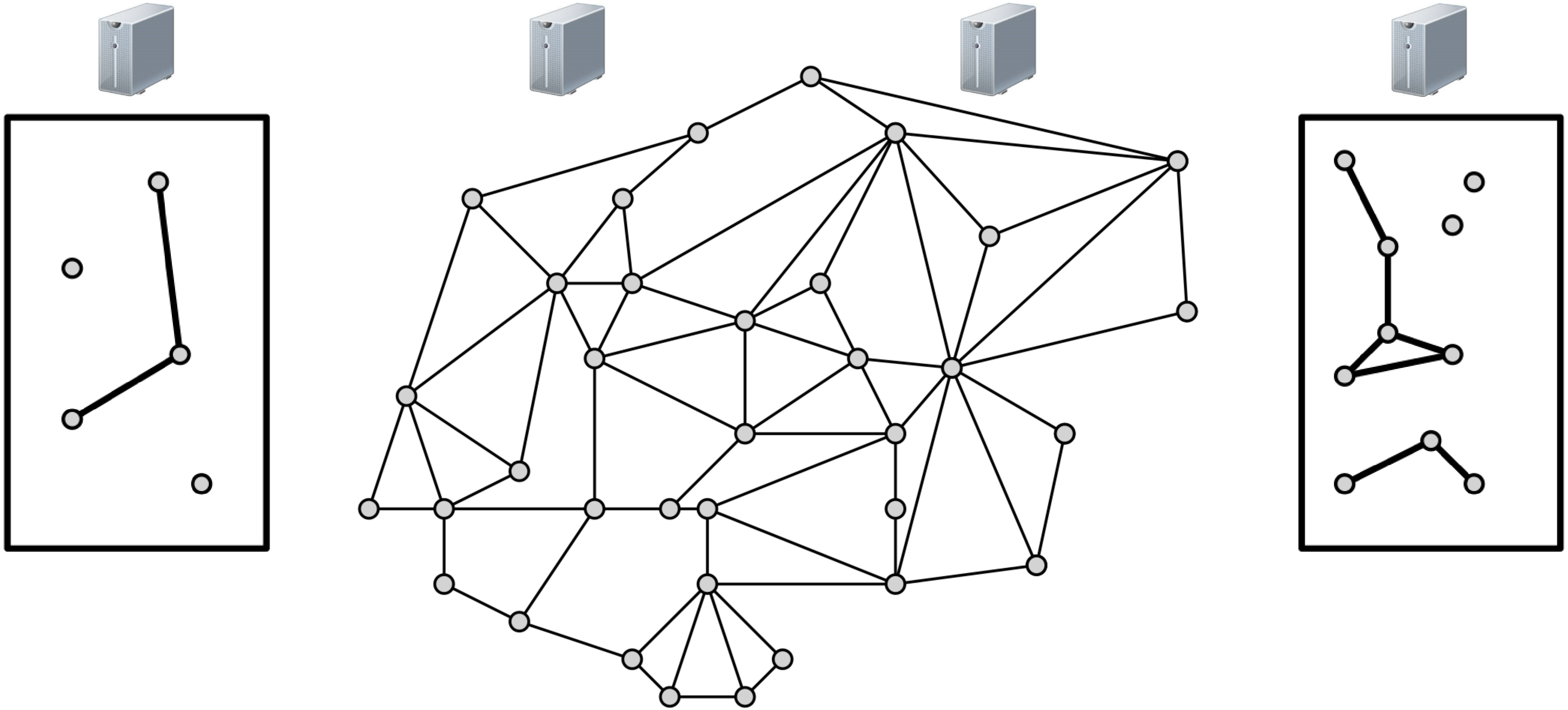


The Plan

- MPC intro
  - LOCAL VS MPC
  - Locality Barrier
- Known Techniques
  - Sparsification and round compression
- New Techniques
  - Total space
  - Careful exponentiation



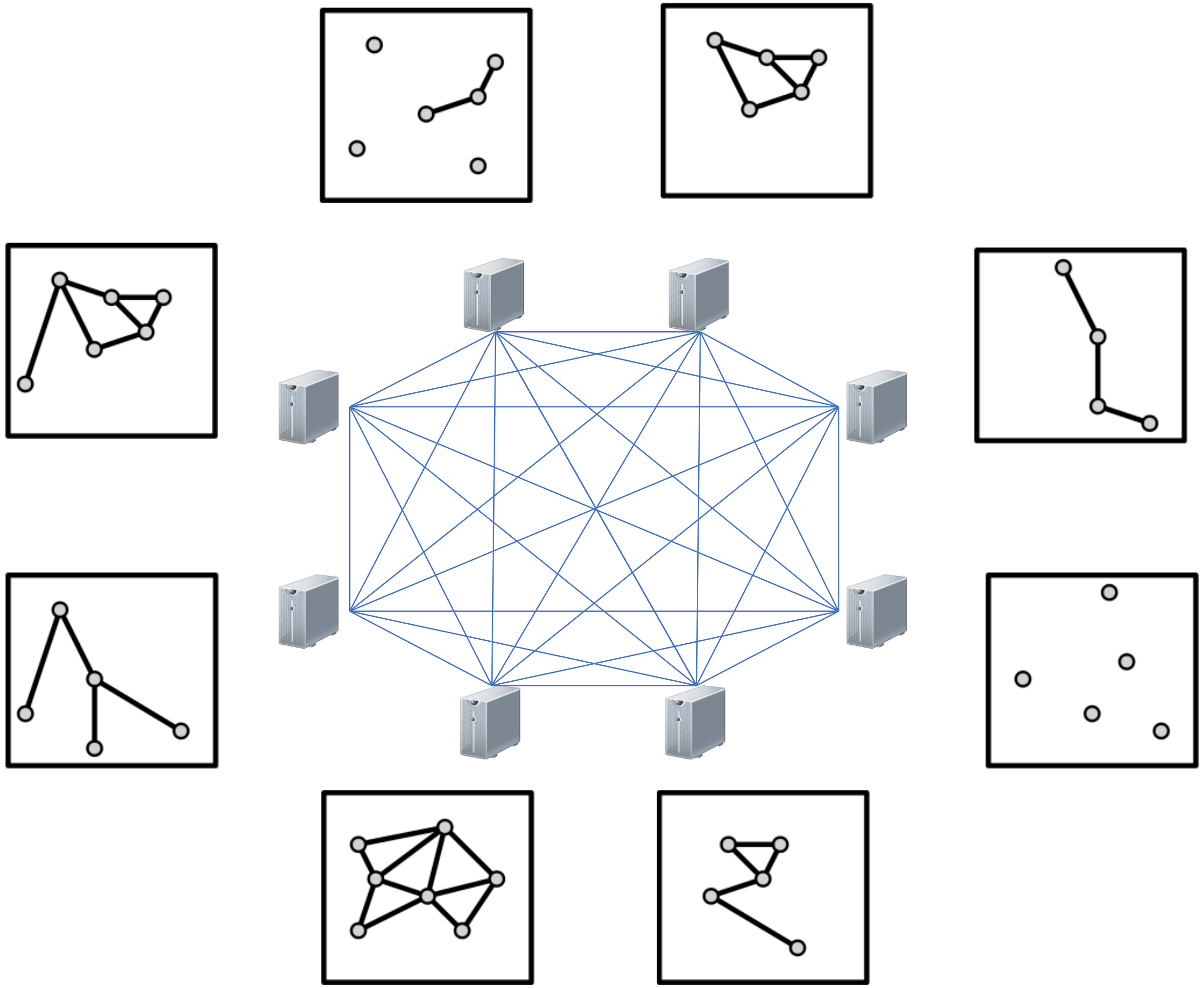
Massively Parallel Computing (MPC)



graph with n nodes and m edges

Massively Parallel Computing (MPC) Model

[Karloff, Suri, Vassilvitskii SODA'10]

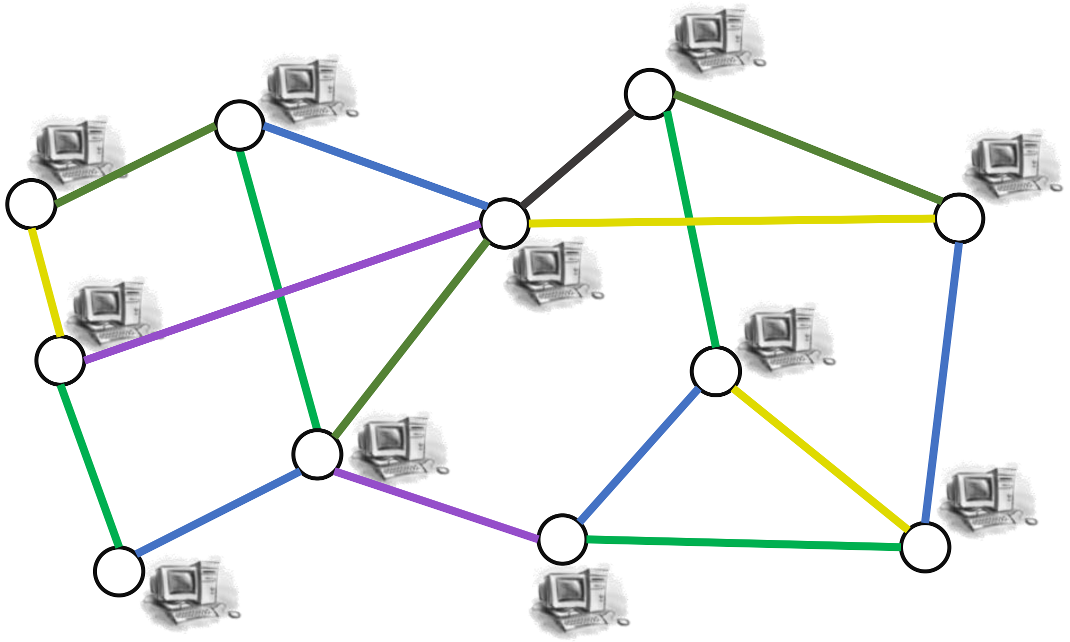


M machines
S memory per machine
Total space $M \cdot S$

Linear space:
$S = \tilde{O}(n)$
A sketch fits onto a single machine

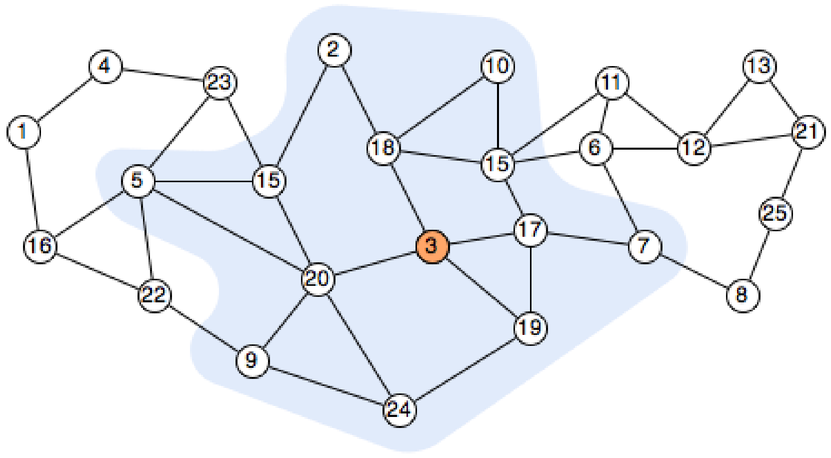
Low-space:
$S = O(n^{\delta}), 0 \leq \delta < 1$
No machine ever sees all the nodes!

MPC vs Message Passing



LOCAL Message Passing
Local algorithms communicate along the edges of the input graph

MPC vs Message Passing



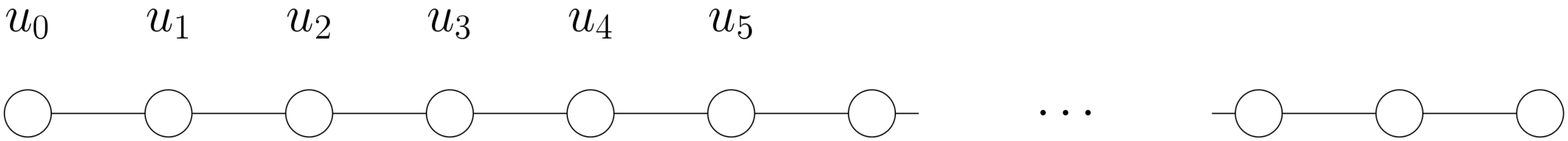
LOCAL:
Map $N^T(v)$ to the output of v

Design pattern in MPC: Graph Exponentiation

Collect the T-hop neighborhoods in $O(\log T)$ rounds.

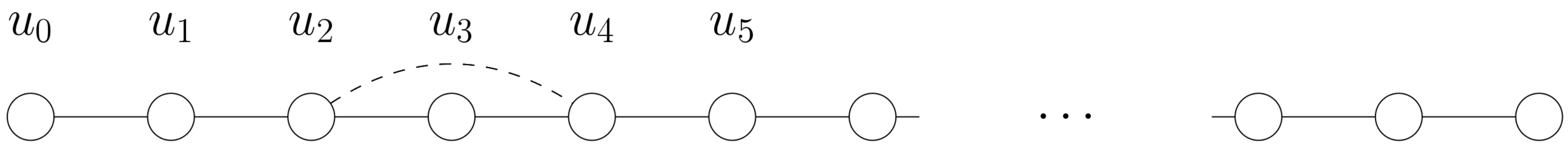
Simulate the LOCAL algorithm.

Graph Exponentiation

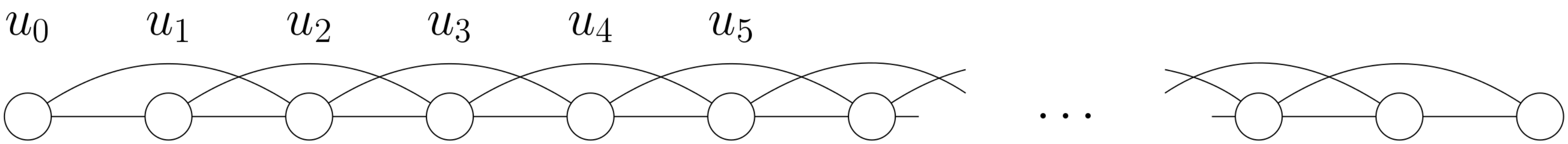


Node u_3 informs its 1-hop neighbors of its 1-hop topology.

Graph Exponentiation

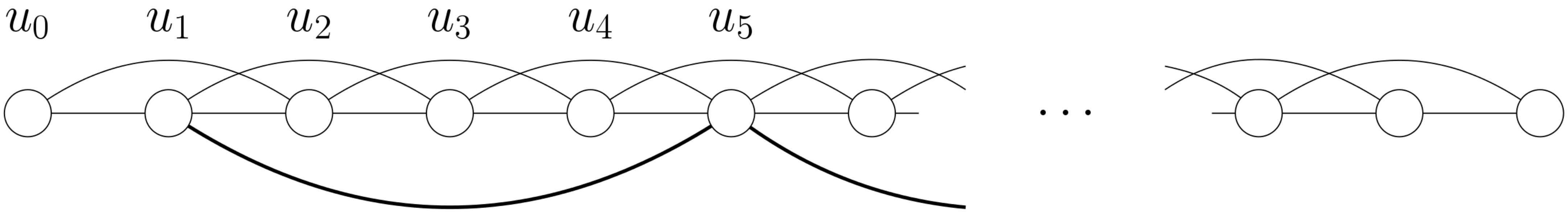


Graph Exponentiation



Everyone learns their 2-hop neighborhood (construct G^2).
Next, communicate 2-hop topology to neighbors in G^2

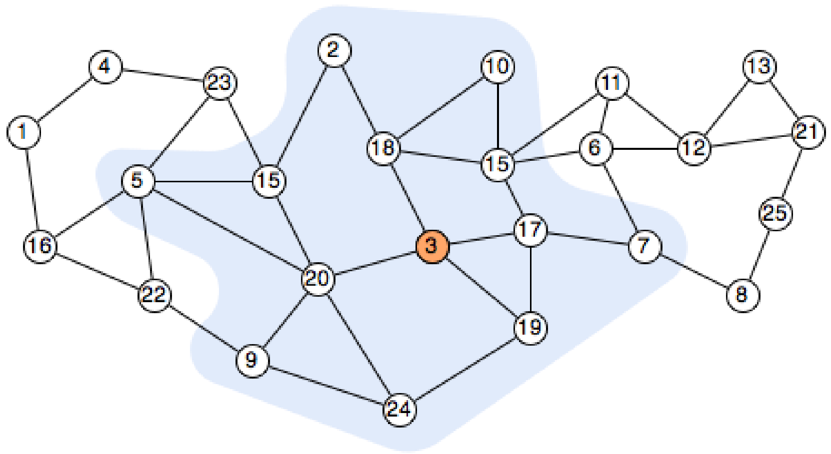
Graph Exponentiation



- By iterating the process, each node v can learn its i-hop neighborhood $N^i(v)$ in $O(\log i)$ MPC rounds.

- We require that $N^i(v)$ fits local memory; $|N^i(v)| \leq n^\delta$

MPC vs Message Passing



Design pattern: **Graph Exponentiation**

Collect the T-hop neighborhoods in $\boldsymbol{O}(\log T)$ rounds.

Simulate the LOCAL algorithm.

<u>$(\Delta + 1)$-coloring</u>

LOCAL: poly $\log\log n$ rounds [GG'23]

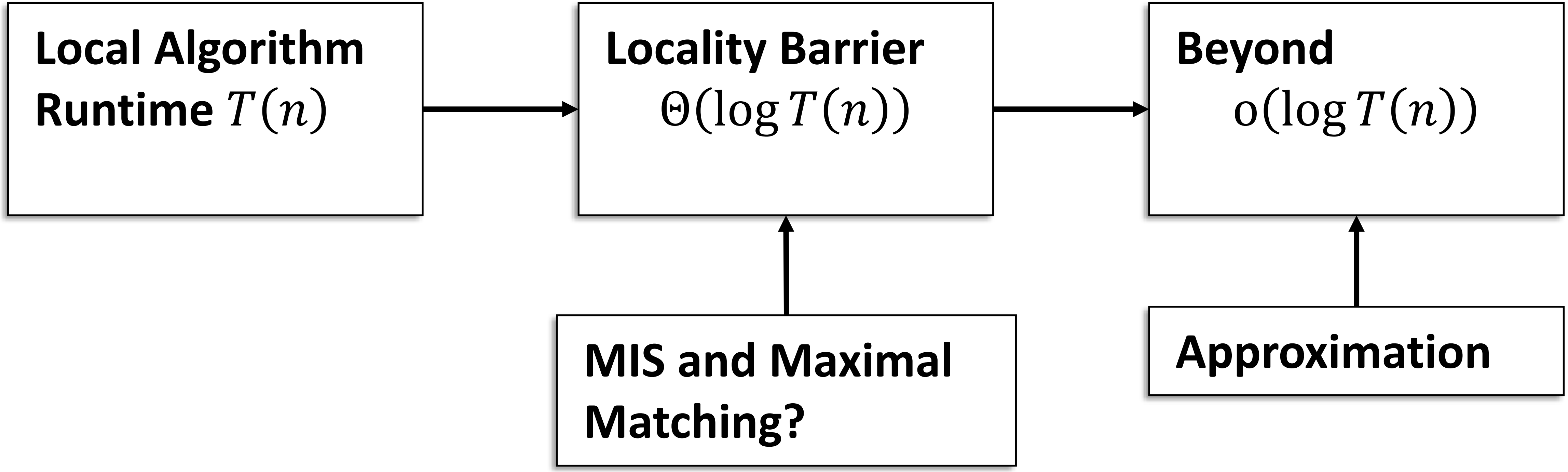
MPC: $O(\log\log\log n)$ rounds [CDP'21]

In some cases, can do even better:
Independent sets of size $\Omega(n/\Delta)$ [KKSS'20, CPD'21]:

LOCAL: $\Omega(\log^* n)$
MPC: $O(1)$

Locality Barrier

Exponentiation gets stuck here!



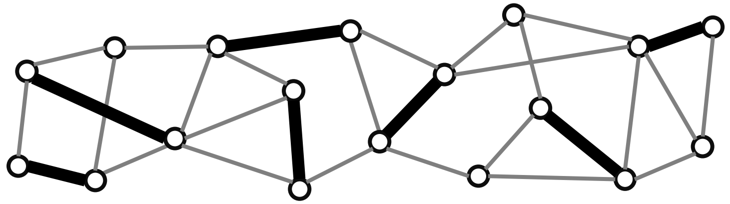
MIS and Maximal Matching

LOCAL [Gh'16]: $O(\log \Delta)$

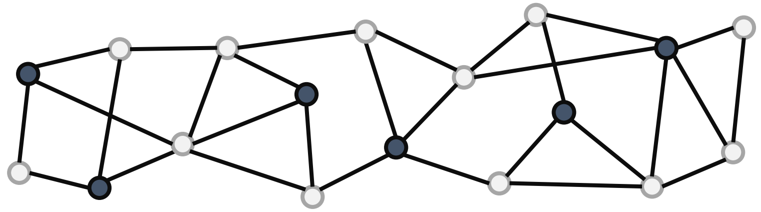
MPC [GU'19]: $\tilde{O}(\sqrt{\log \Delta})$

??

Locality barrier: $\Theta(\log \log \Delta)$

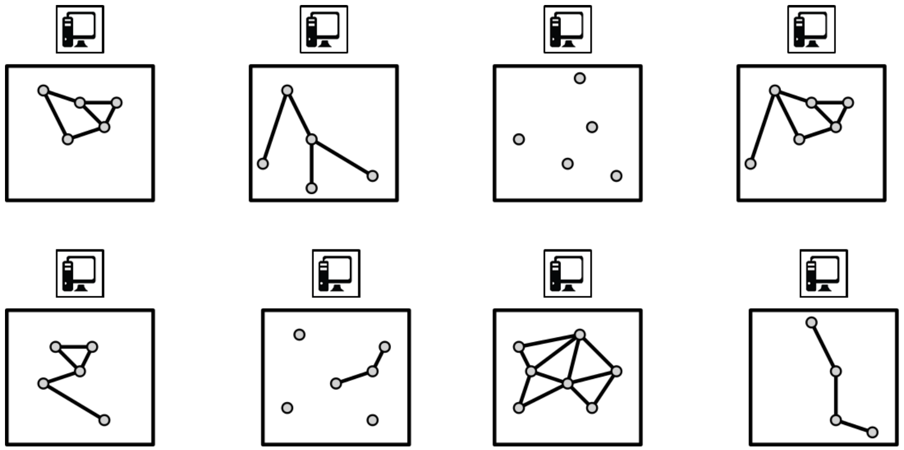


Maximal Matching



Maximal Independent Set

The Plan



- MPC intro
  - LOCAL VS MPC
  - Locality Barrier
- Known Techniques
  - Sparsification and round compression
- New Techniques
  - Total space
  - Careful exponentiation

Round Compression and Sparsification

MIS Sparsification Simplified (a lot):

1. Consider Ghaffari's algorithm that runs in $T = O(\log \Delta)$ rounds.
2. Simulate the algorithm on a sparse (low degree) subgraph for $\Omega\left(\sqrt{\log \Delta}\right)$ rounds.
3. Repeat $O\left(\sqrt{\log \Delta}\right)$ times.

Degree $d = 2^{\sqrt{\log \Delta}}$ and $d^{\sqrt{\log \Delta}} \leq \Delta \leq S$

$O(\sqrt{\log \Delta} \cdot \log \log \Delta)$ rounds in total.

Round Compression and Sparsification

Seems like a fundamental barrier

MIS Sparsification Simplified (a lot):

1. Consider Ghaffari's algorithm that runs in $T = O(\log \Delta)$ rounds.
2. Simulate the algorithm on a sparse (low degree) subgraph for $\Omega\left(\sqrt{\log \Delta}\right)$ rounds.
3. Repeat $O\left(\sqrt{\log \Delta}\right)$ times.

Black box application of exponentiation.

$O(\sqrt{\log \Delta} \cdot \log \log \Delta)$ rounds in total.

Shattering

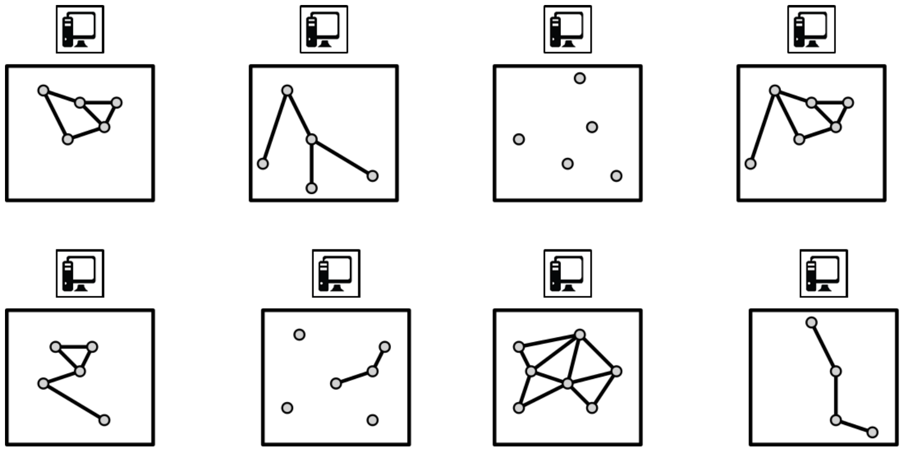
MIS Sparsification Simplified (a lot):
After $O\left(\sqrt{\log \Delta}\right)$ iterations, the graph *shatters* into $O(\log n)$ sized components.

Post-shattering in MPC: gather the small components and simulate LOCAL.

Black box application of exponentiation.

The Plan

- MPC intro
  - LOCAL VS MPC
  - Locality Barrier
- Known Techniques
  - Sparsification and round compression
- New Techniques
  - Total space
  - Careful exponentiation



What to Do?

Locally Checkable Problems:
Almost all approaches rely, to some extent, on black-box exponentiation.

If each node gathers its t-hop neighborhood of size B, we need $O(B \cdot n)$ total space.

How to change the game?

Limit total space to linear?

MIS:

Smarter use of the sparsified graph?

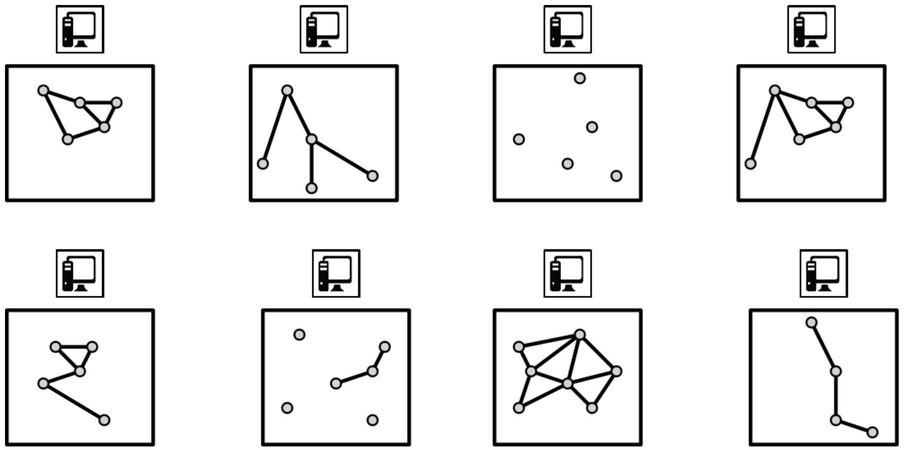
What to Do?

At the least:
Come up with new ideas and algorithms.

Probably:
Learn ways to collect local data fast

The Plan

- MPC intro
  - LOCAL VS MPC
  - Locality Barrier
- Known Techniques
  - Sparsification and round compression
- New Techniques
  - Total space
  - Careful exponentiation



LCLs in the “Tiny” Regime

Theorem [CKP’19]:

Any LCL with deterministic locality $o(\log n)$ can be solved with a canonical (LOCAL) algorithm in $O(\log^* n)$ rounds.

Need a distance-k coloring

Get it through coloring of G^k in $O(\log^* n)$ rounds of LOCAL

Locally Checkable Labeling (LCL):

1. Solution can be checked locally
2. Constant degree graphs

Linial’s Algorithm:
In one round, turn a c-coloring into $O(\Delta^2 \log c)$-coloring.

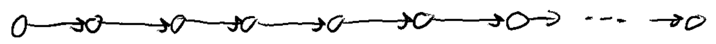
Coloring Pseudo-Forests

Coloring of G^k

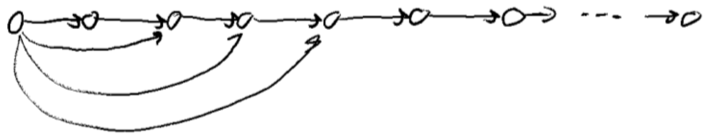
Since Δ and k are constants, can focus on 3-coloring pseudo-forests.
Linial: enough to look at $O(\log^* n)$ ancestors.

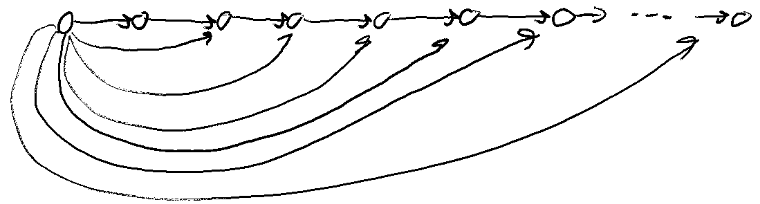
Important: Focus on MPC issues.

A tempting approach:
Gather $O(\log^* n)$-neighborhood



Requires $\Omega(n \log^* n)$ total space!





Coloring a Directed Pseudo-Forest

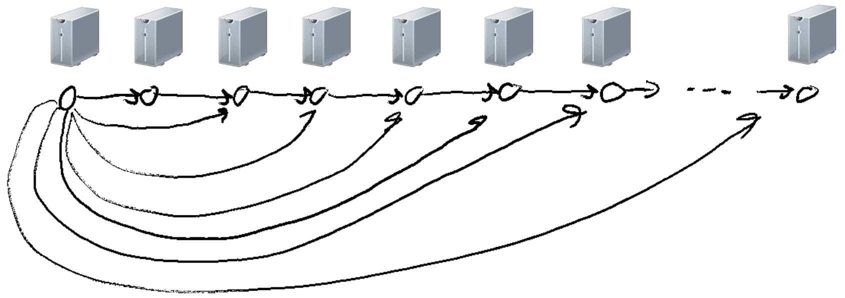
Careful Exploration

Run just one round of Linial's
 - Turn $\log n$ bit IDs into $\log \log n$ bit colors

Collect a vector of size $O(\log \log n \cdot \log^* n) = O(\log n)$ bits

Total space: $O(n)$ words.

Issue:
Need to store $O(\log^* n)$ machine addresses of $\Omega(\log n)$ bits.



Coloring a Directed Pseudo-Forest

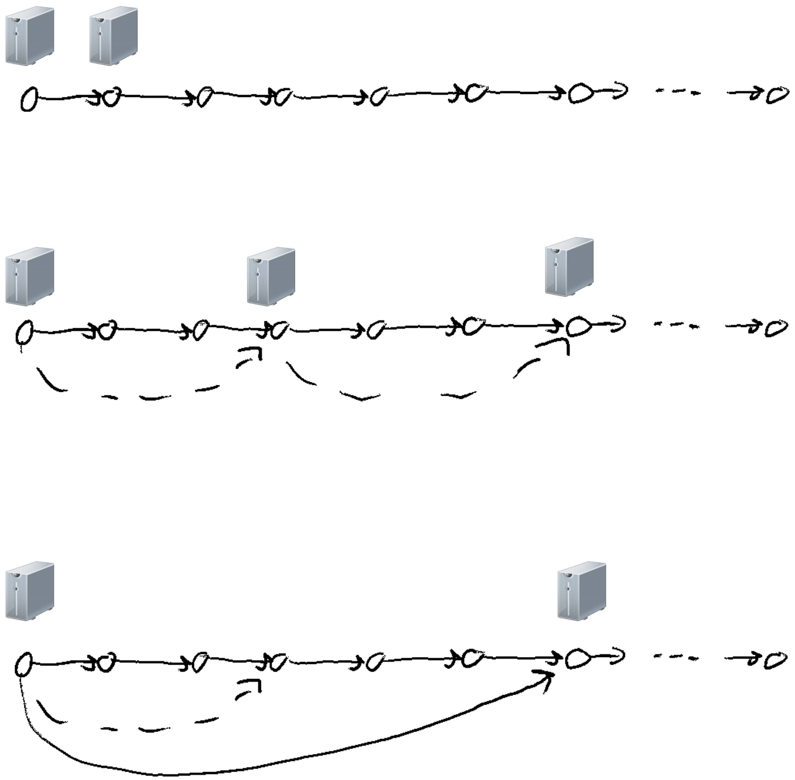
Careful Exploration

Run just one round of Linial's
- Turn IDs into $\log\log n$ -bit colors

Collect a vector of size $O(\log\log n \cdot \log^* n) = O(\log n)$ bits

Only store the address of farthest machine, $O(\log n)$ bits.

Total space: $O(n)$ words.



LCLs in the "Tiny" Regime

Theorem [BBFLMOU'20]

For any LCL P with locality $o(\log n)$, there is an MPC algorithm that solves P in $O(\log \log^* n)$ rounds.

Nice property:
Optimal in terms of memory parameters.

Nice property:
Goes beyond naïve exponentiation.

Nice property:
Runtime potentially optimal.

Chicken vs Egg

Is there a difference between (?)

1. First creating a smart subgraph and doing naïve exponentiation

2. Smart exponentiation on the input graph