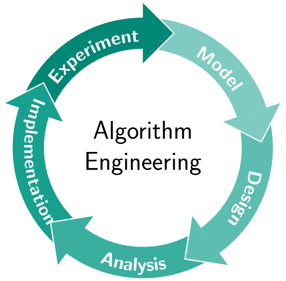
Engineering Scalable Distributed List Ranking

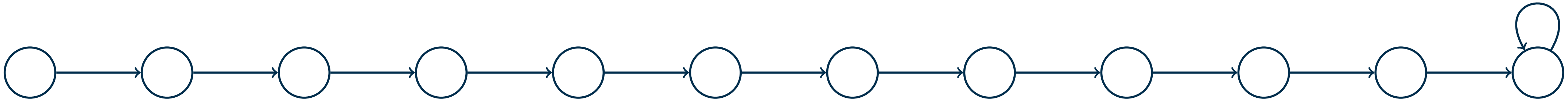


Euro-Par 2026, Pisa | August 27, 2026

Peter Sanders, Matthias Schimek, **Tim Niklas Uhl**, Thomas Weidmann

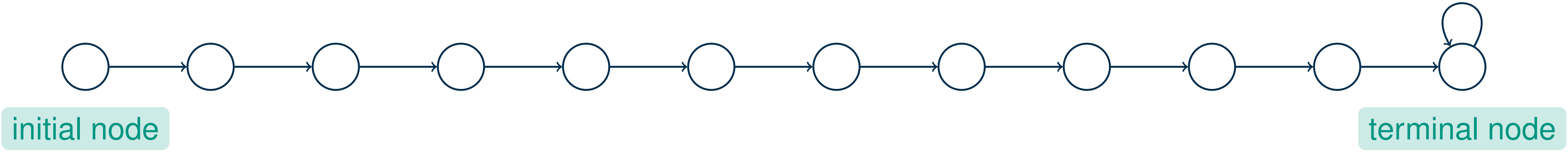
The List Ranking Problem

A Parallel Algorithms Textbook Classic



- **"fruit fly problem"**: simple problem, same patterns as other graph algorithms
- extensively researched (in theory and practice) in the 90s, not much work since
- we want to **revisit it on modern distributed systems**
- **Applications:** list prefix sums, convert lists to arrays, operations on trees (via Euler tours), chain compaction in genome assembly

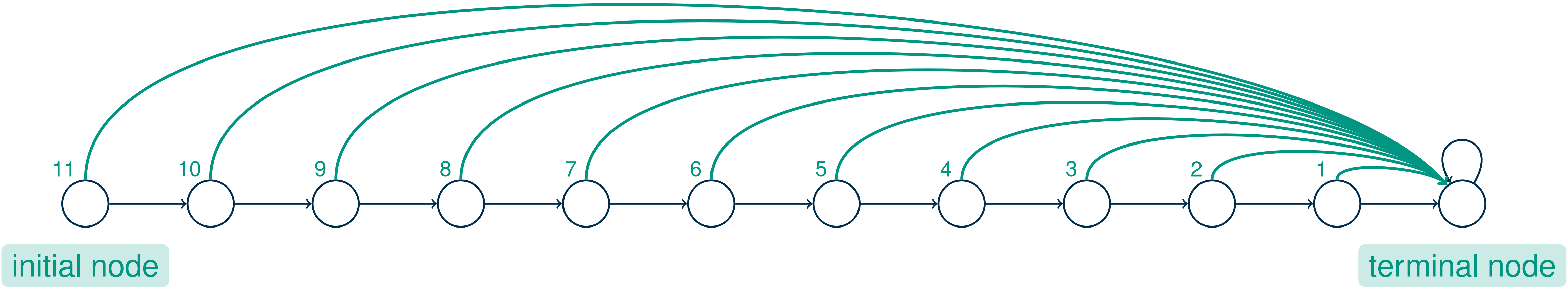
The List Ranking Problem
A Parallel Algorithms Textbook Classic



- **"fruit fly problem"**: simple problem, same patterns as other graph algorithms
- extensively researched (in theory and practice) in the 90s, not much work since
- we want to **revisit it on modern distributed systems**
- **Applications:** list prefix sums, convert lists to arrays, operations on trees (via Euler tours), chain compaction in genome assembly

The List Ranking Problem

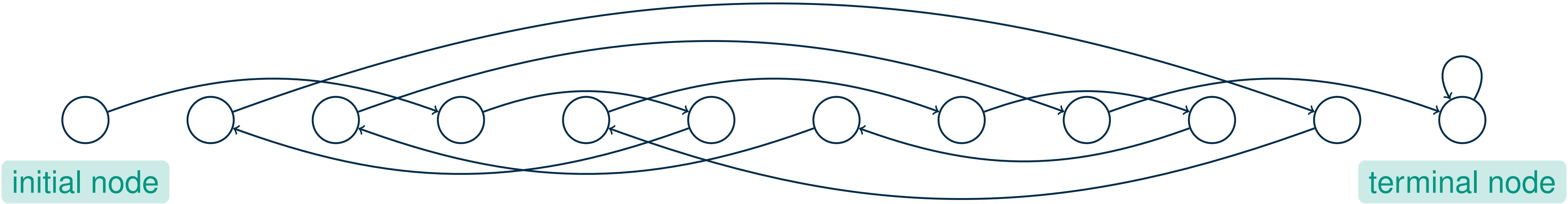
A Parallel Algorithms Textbook Classic



- **"fruit fly problem"**: simple problem, same patterns as other graph algorithms
- extensively researched (in theory and practice) in the 90s, not much work since
- we want to **revisit it on modern distributed systems**
- **Applications:** list prefix sums, convert lists to arrays, operations on trees (via Euler tours), chain compaction in genome assembly

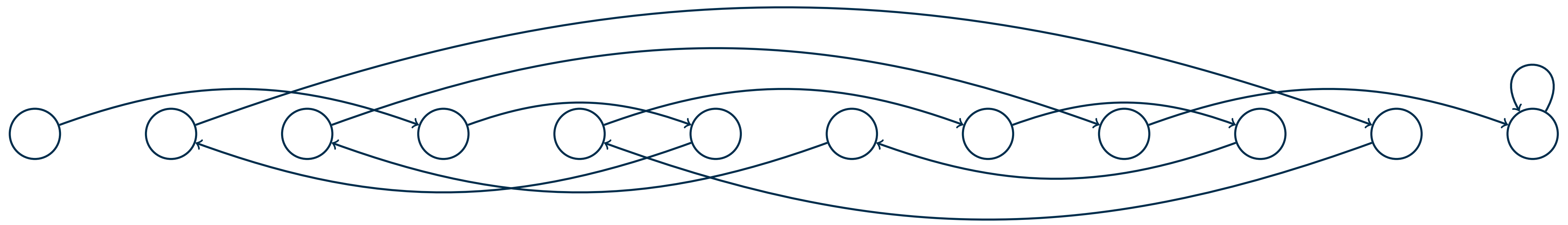
The List Ranking Problem

A Parallel Algorithms Textbook Classic



The List Ranking Problem

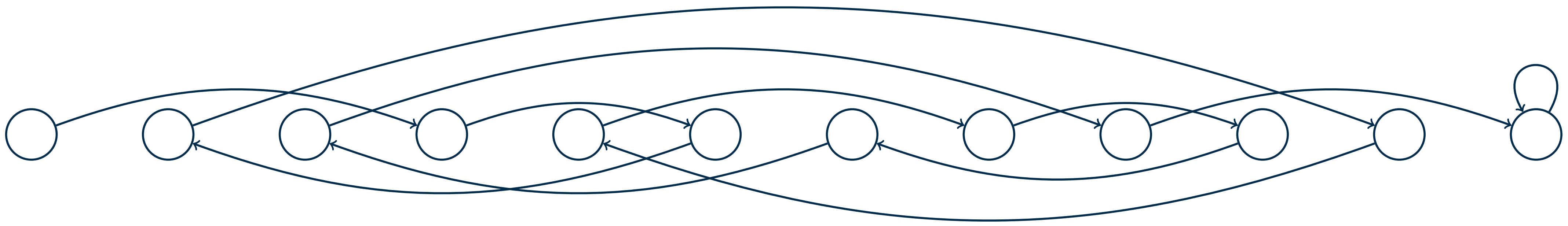
A Parallel Algorithms Textbook Classic



	0	1	2	3	4	5	6	7	8	9	10	11
succ	3	10	8	5	7	1	2	9	11	6	4	11
rank	–	–	–	–	–	–	–	–	–	–	–	0

The List Ranking Problem

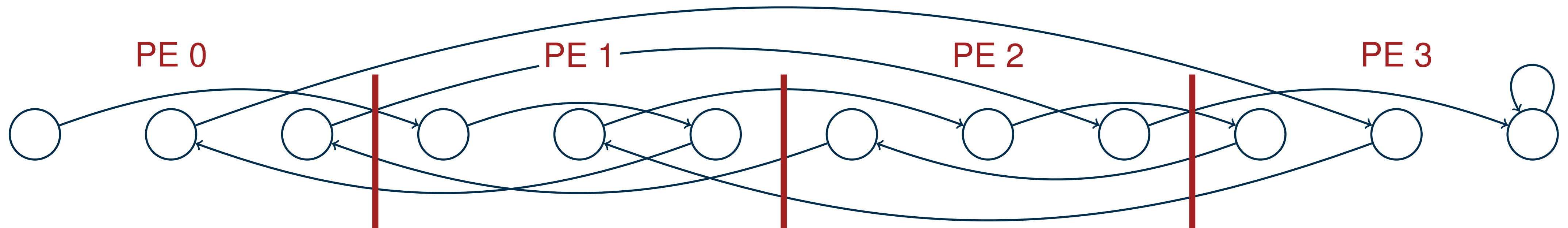
A Parallel Algorithms Textbook Classic



	0	1	2	3	4	5	6	7	8	9	10	11
succ	3	10	8	5	7	1	2	9	11	6	4	11
rank	11	8	2	10	6	9	3	5	1	4	7	0

The List Ranking Problem

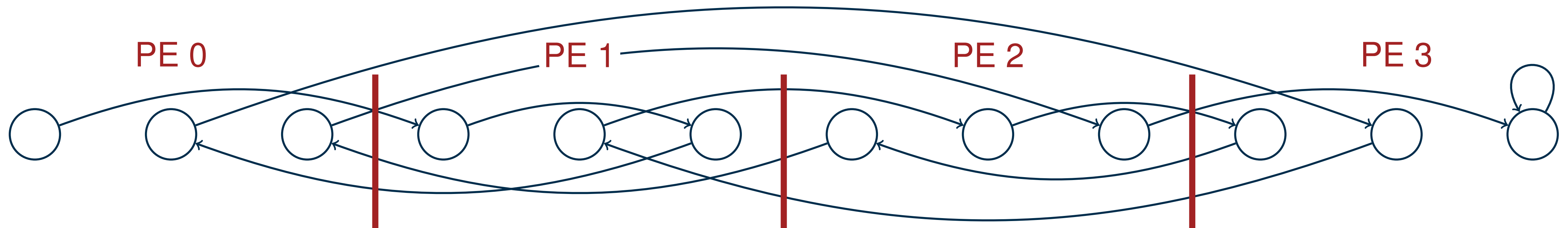
A Parallel Algorithms Textbook Classic



	0	1	2	3	4	5	6	7	8	9	10	11
succ	3	10	8	5	7	1	2	9	11	6	4	11
rank	11	8	2	10	6	9	3	5	1	4	7	0

The List Ranking Problem

A Parallel Algorithms Textbook Classic

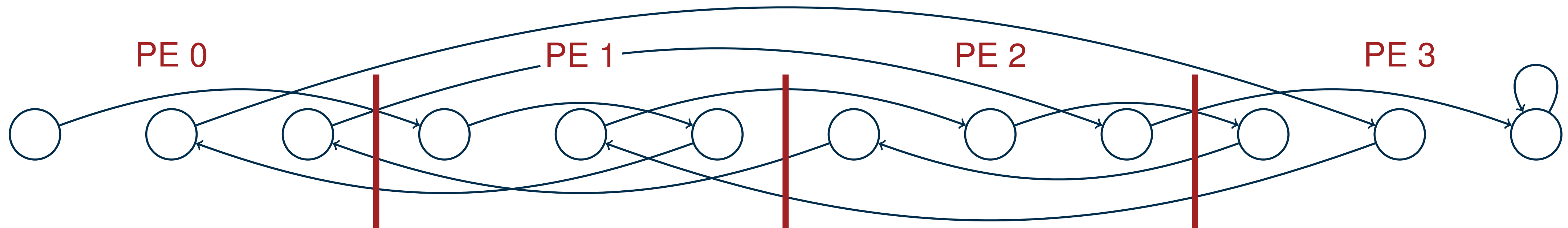


	0	1	2	3	4	5	6	7	8	9	10	11
succ	3	10	8	5	7	1	2	9	11	6	4	11
rank	11	8	2	10	6	9	3	5	1	4	7	0

$\approx n/p$

The List Ranking Problem

A Parallel Algorithms Textbook Classic



	0	1	2	3	4	5	6	7	8	9	10	11
succ	3	10	8	5	7	1	2	9	11	6	4	11
rank	11	8	2	10	6	9	3	5	1	4	7	0

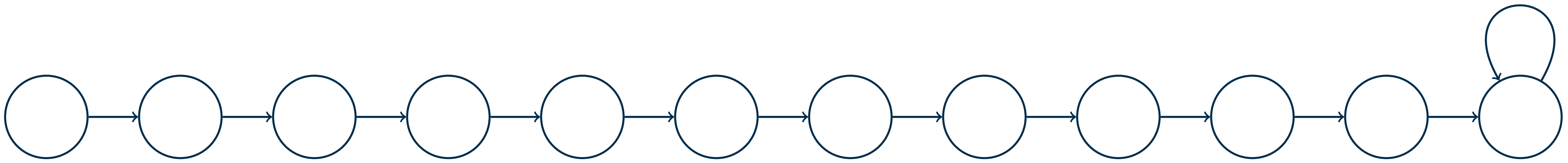
$\approx n/p$

sending ℓ words takes time

$$\alpha + \beta\ell$$

Pointer Doubling

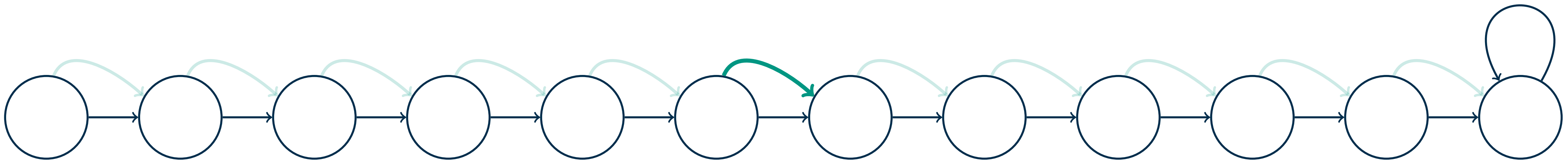
One of the First PRAM Algorithms (Wyllie 1979)



- $\mathcal{O}(\log n)$ rounds, each a bulk-synchronous **all-to-all exchange**
- not work-efficient

Pointer Doubling

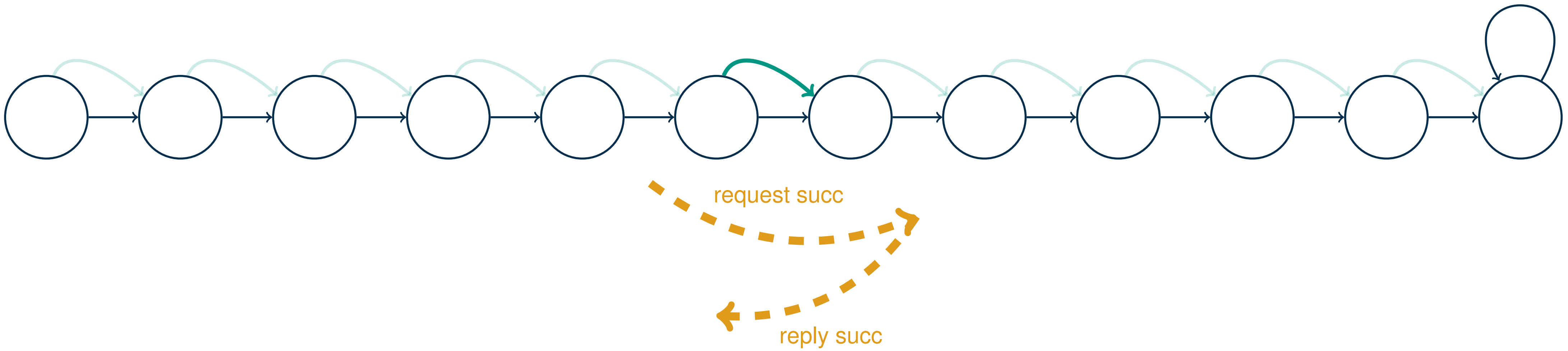
One of the First PRAM Algorithms (Wyllie 1979)



- $\mathcal{O}(\log n)$ rounds, each a bulk-synchronous **all-to-all exchange**
- not work-efficient

Pointer Doubling

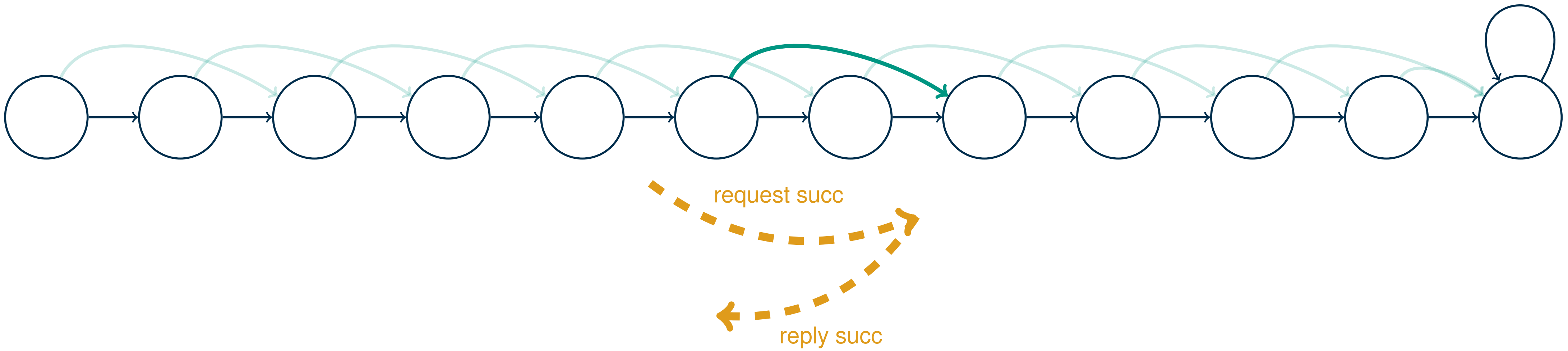
One of the First PRAM Algorithms (Wyllie 1979)



- $\mathcal{O}(\log n)$ rounds, each a bulk-synchronous **all-to-all exchange**
- not work-efficient

Pointer Doubling

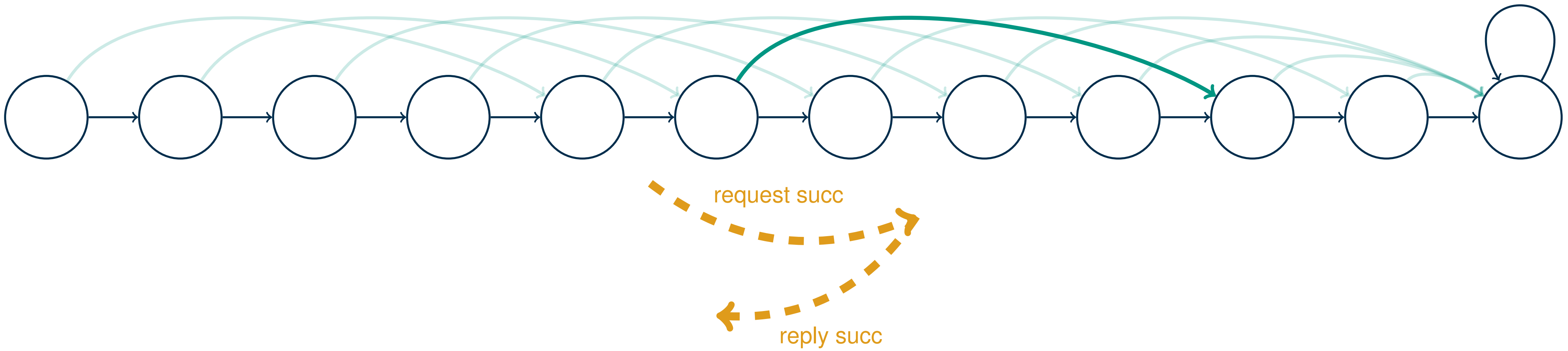
One of the First PRAM Algorithms (Wyllie 1979)



- $\mathcal{O}(\log n)$ rounds, each a bulk-synchronous **all-to-all exchange**
- not work-efficient

Pointer Doubling

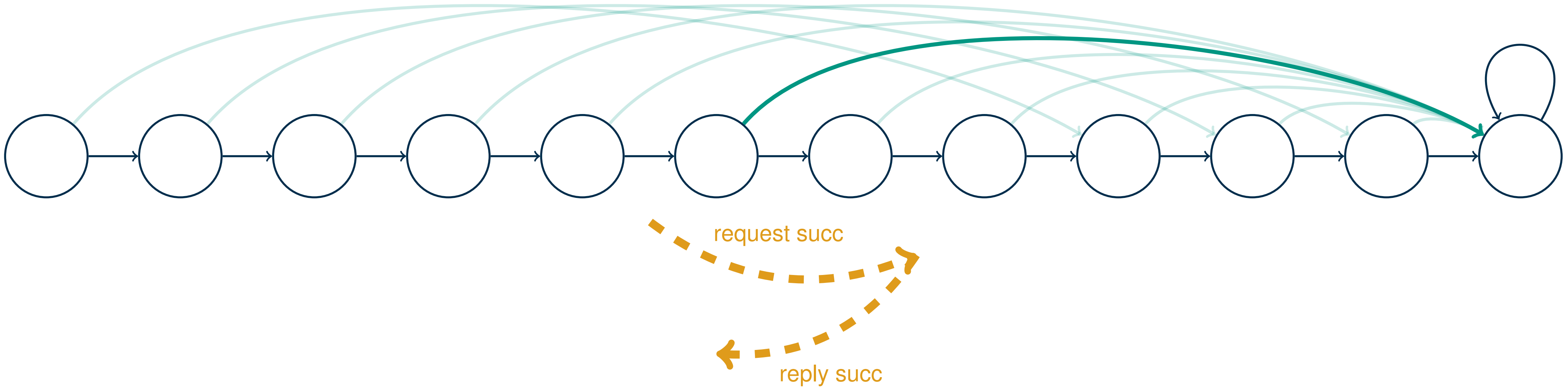
One of the First PRAM Algorithms (Wyllie 1979)



- $\mathcal{O}(\log n)$ rounds, each a bulk-synchronous **all-to-all exchange**
- not work-efficient

Pointer Doubling

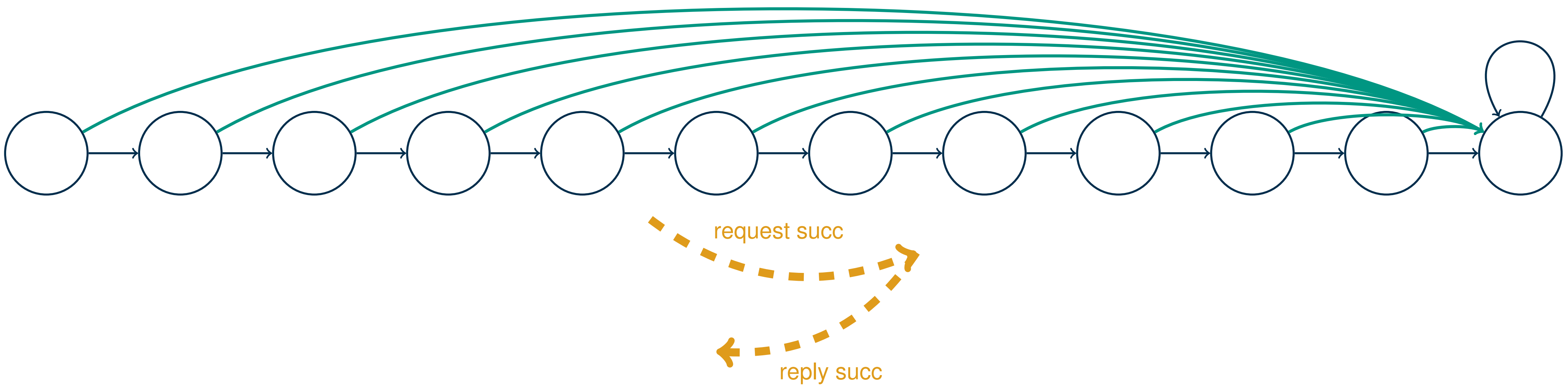
One of the First PRAM Algorithms (Wyllie 1979)



- $\mathcal{O}(\log n)$ rounds, each a bulk-synchronous **all-to-all exchange**
- not work-efficient

Pointer Doubling

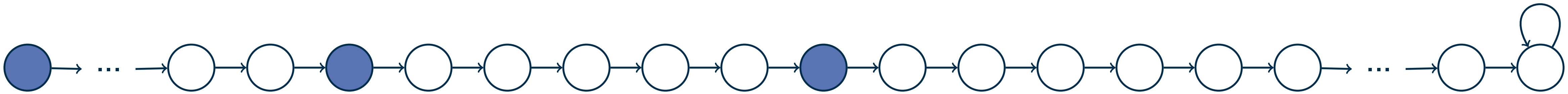
One of the First PRAM Algorithms (Wyllie 1979)



- $\mathcal{O}(\log n)$ rounds, each a bulk-synchronous **all-to-all exchange**
- not work-efficient

Sparse Ruling Set

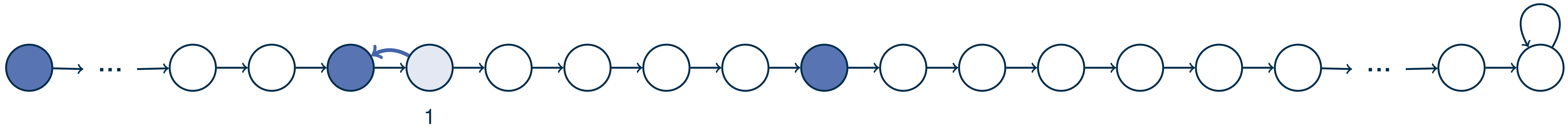
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

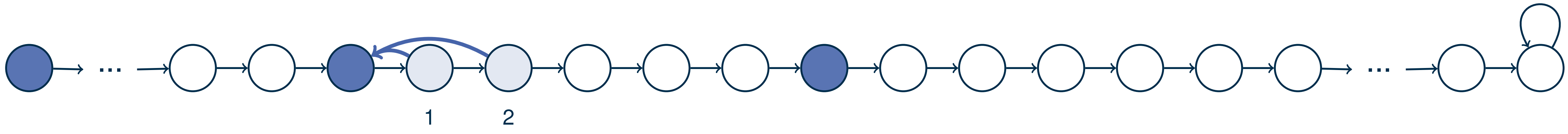
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

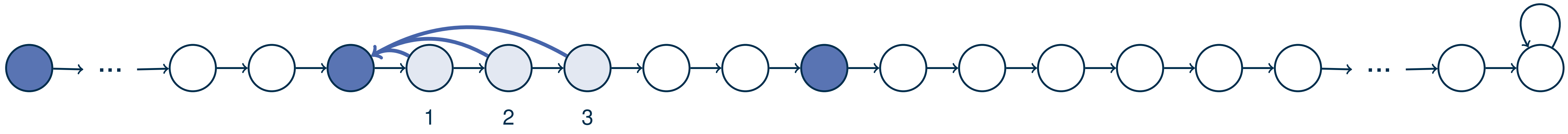
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

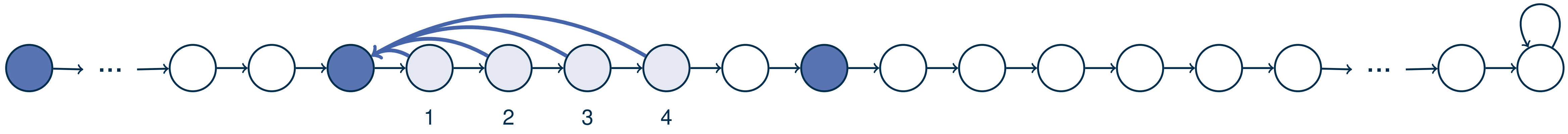
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

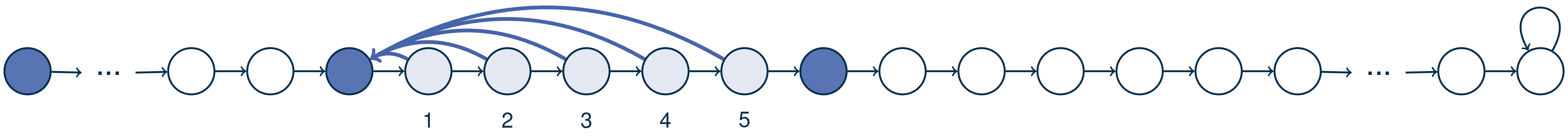
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

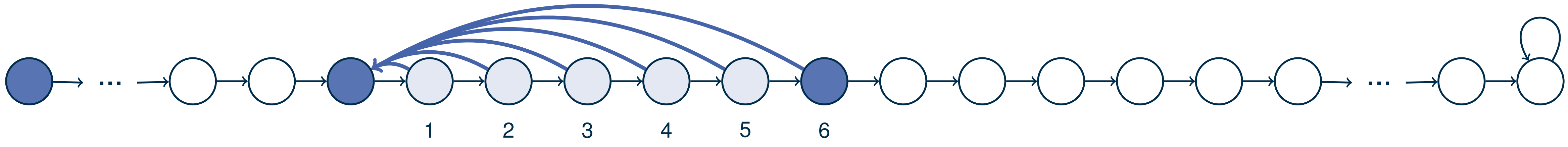
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

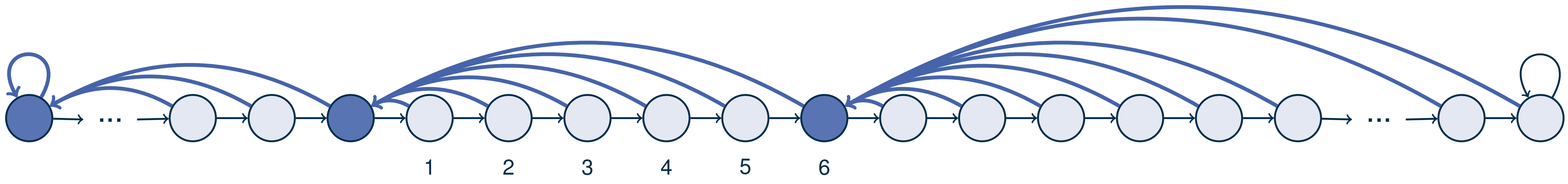
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

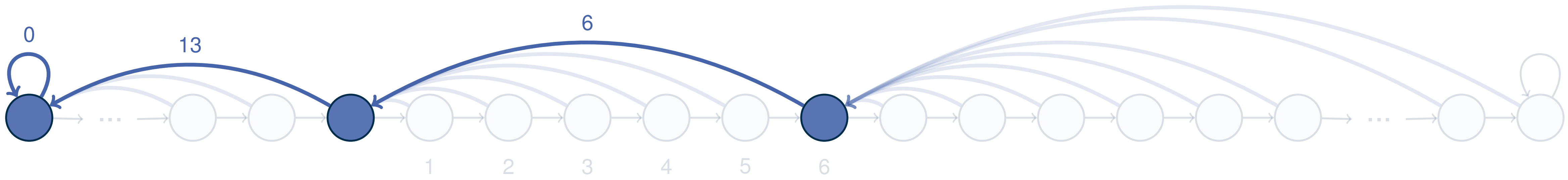
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

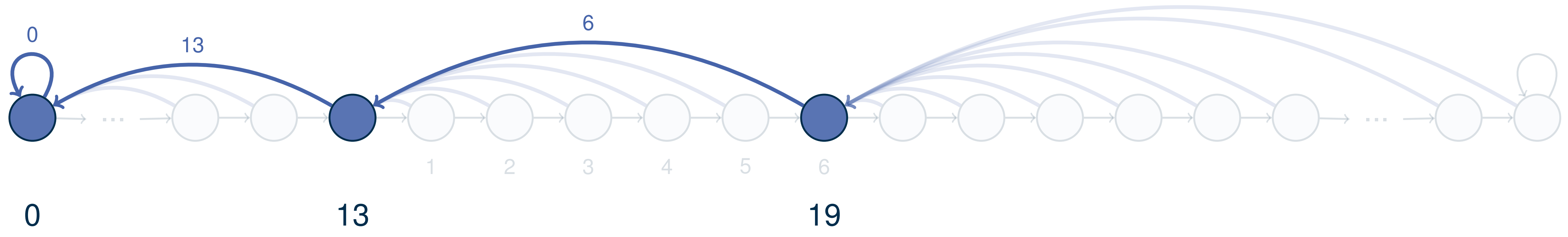
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

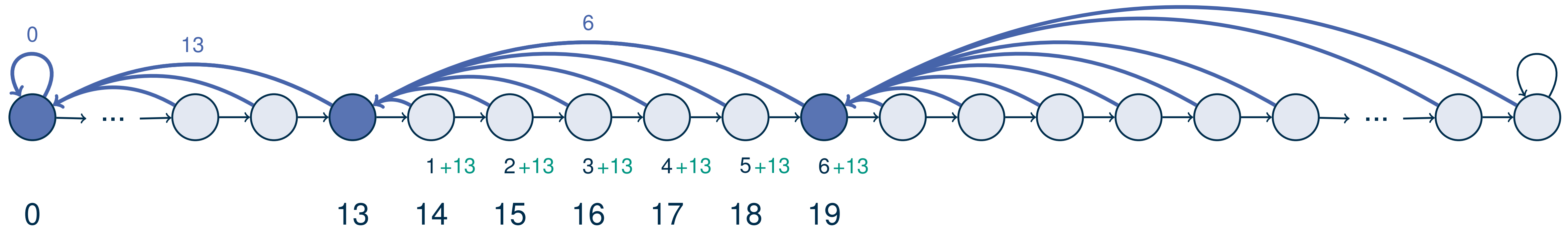
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

Sparse Ruling Set

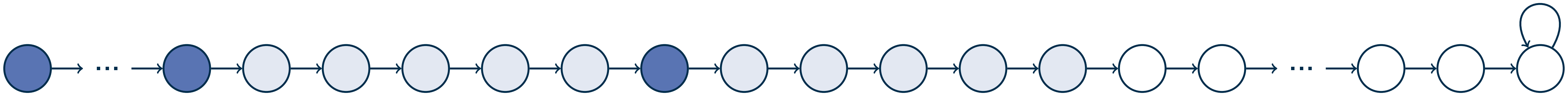
The Base Algorithm (Reid-Miller 1994; Anderson and Miller 1991)



- select random set $\mathcal{R}$ of **rulers**
- traverse sub-lists until we reach a ruler (**ruler chasing**)
- compress segments between rulers
- rank sub-problem (**base-case**)
- compute final ranks and terminals (**ruler propagation**)

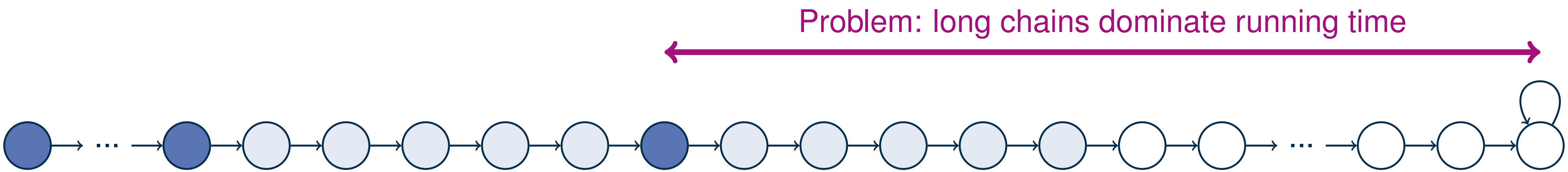
Sparse Ruling Set

Ruler Spawning (Sibeyn 1999)



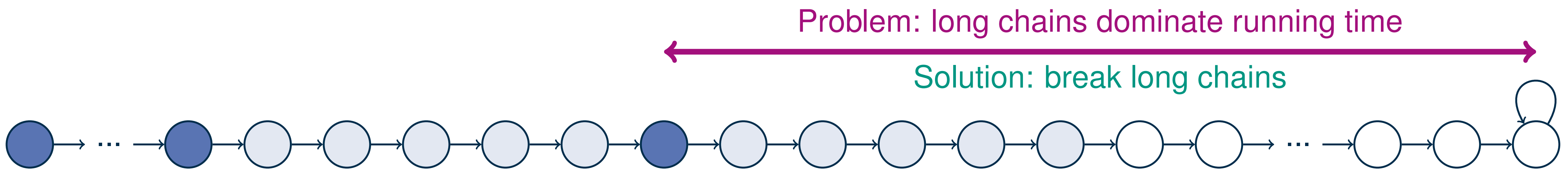
Sparse Ruling Set

Ruler Spawning (Sibeyn 1999)



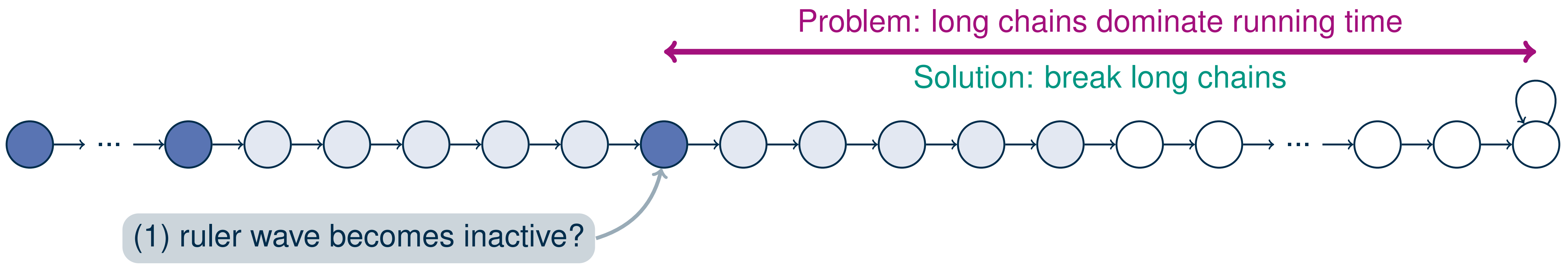
Sparse Ruling Set

Ruler Spawning (Sibeyn 1999)



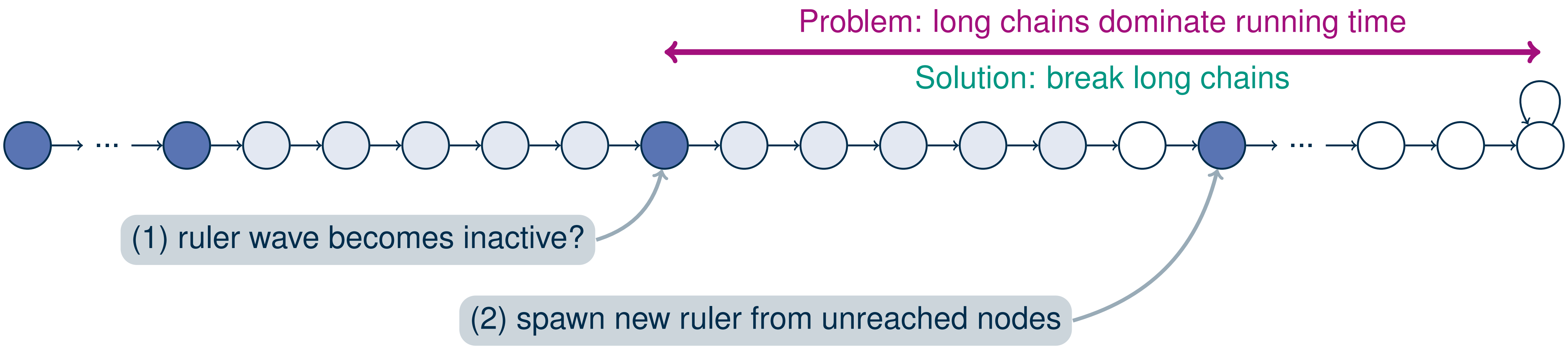
Sparse Ruling Set

Ruler Spawning (Sibeyn 1999)



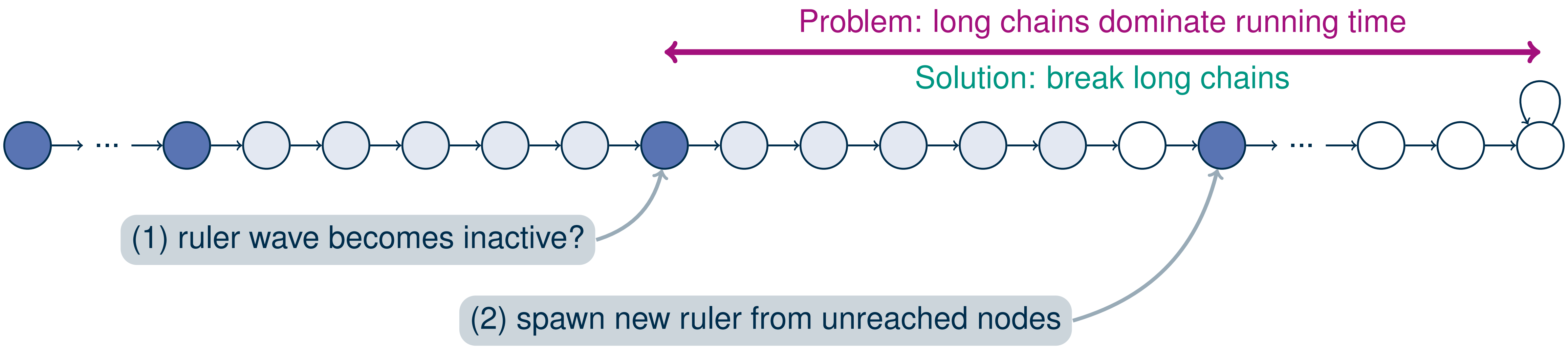
Sparse Ruling Set

Ruler Spawning (Sibeyn 1999)



Sparse Ruling Set

Ruler Spawning (Sibeyn 1999)



- $|\mathcal{R}| =: r$ ruler waves active at any time
- at most n/r communication rounds
- sub-problem becomes $\ln(n/r)$ times larger
- **open question:** how to choose rulers?

Making it Scale

Engineering Scalable List Ranking

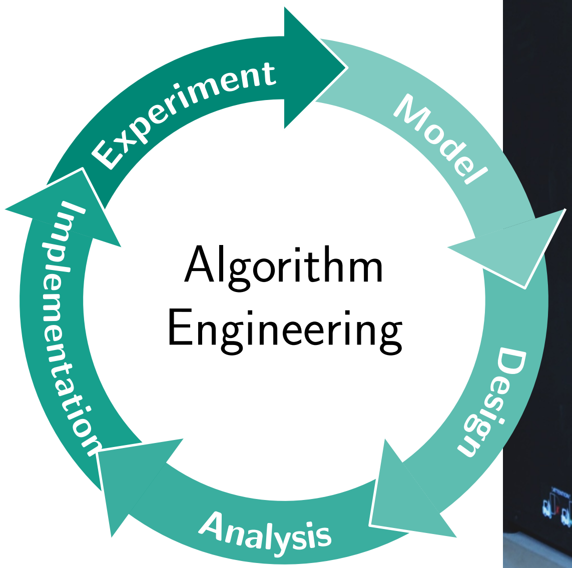
- prior practical work on SRS topped out at $\approx$ 130 **PEs** (Sibeyn et al. 1999)
- **our target:** SuperMUC-NG @ LRZ
  - we use **up to 24 576 cores**
- requires new engineering: **locality awareness** and **reducing latency**

C++23, built with

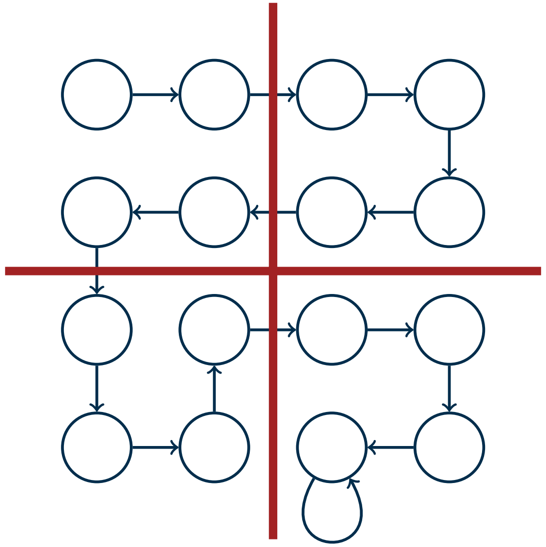
KaMPIng

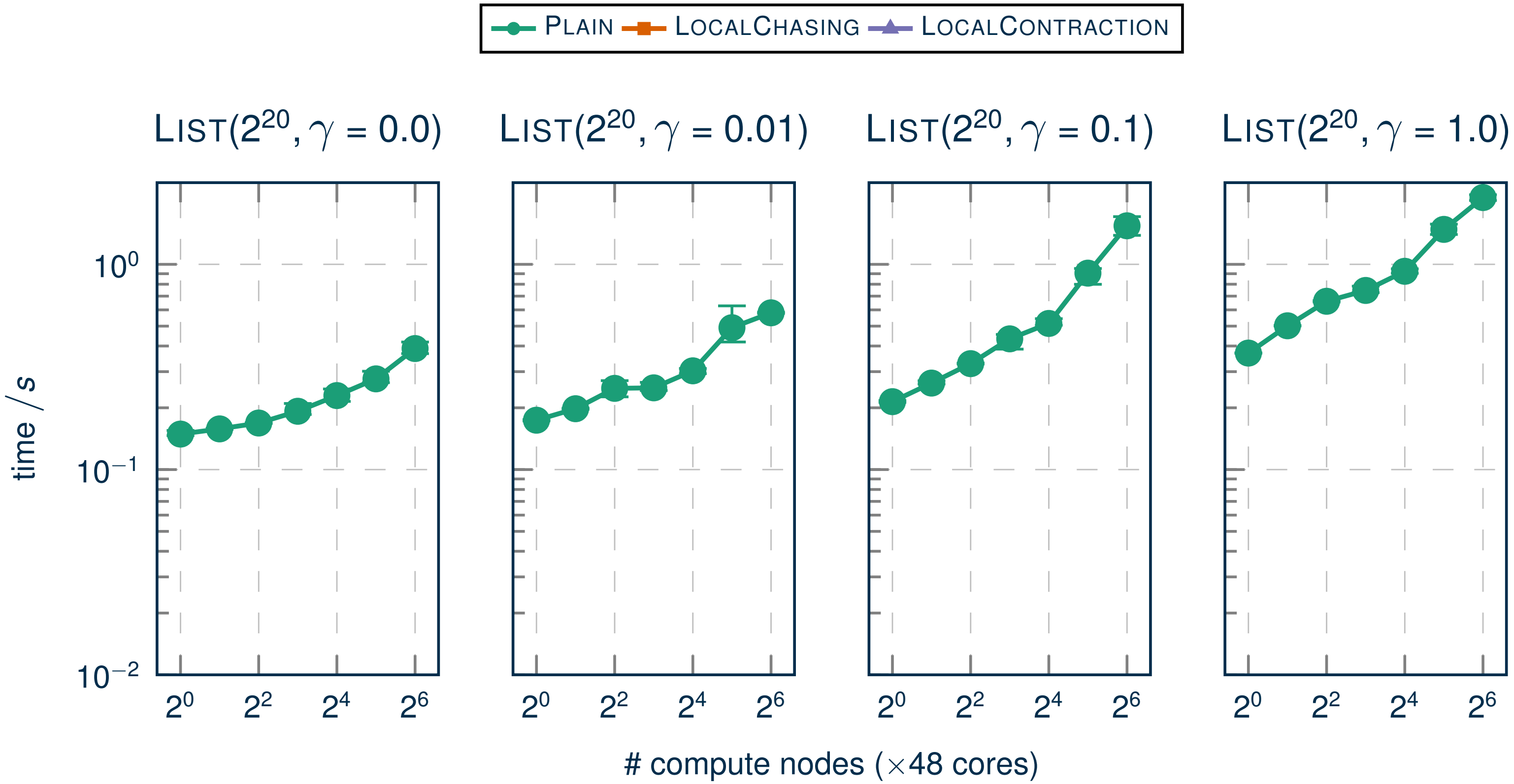
Open Source

niklas-uhl/kascade



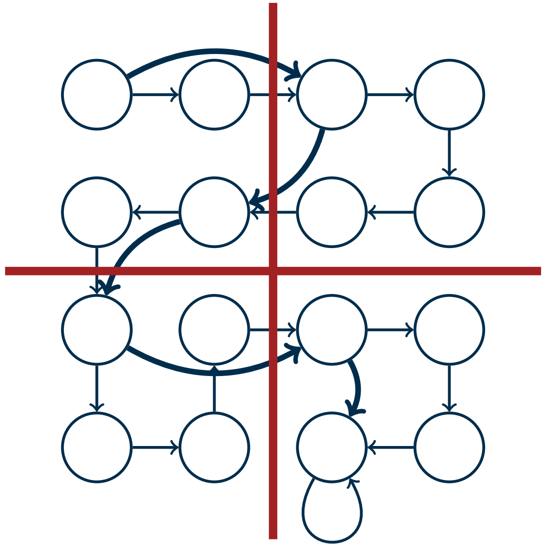
Making it Scale
Exploiting Locality

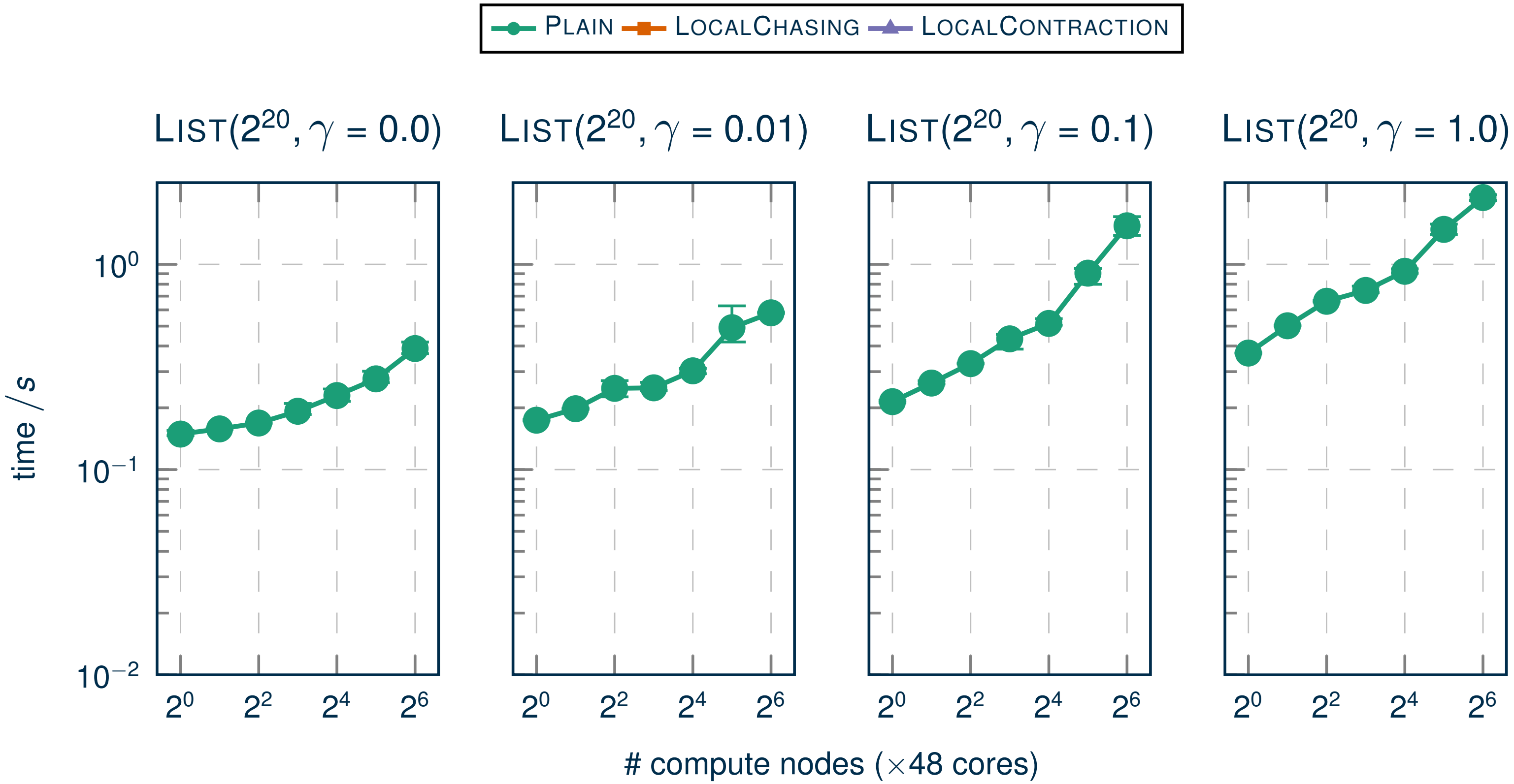




Making it Scale

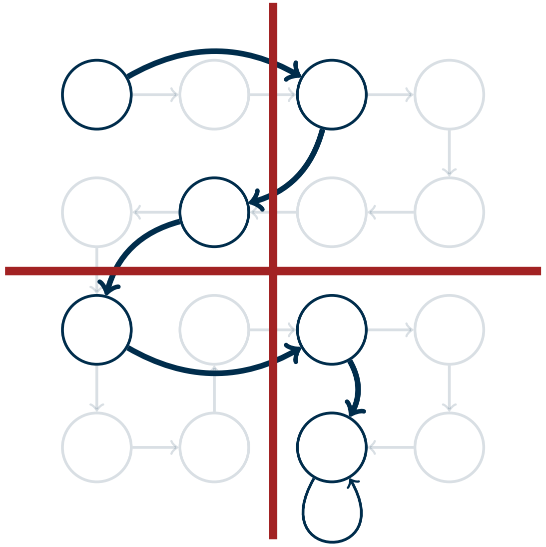
Exploiting Locality

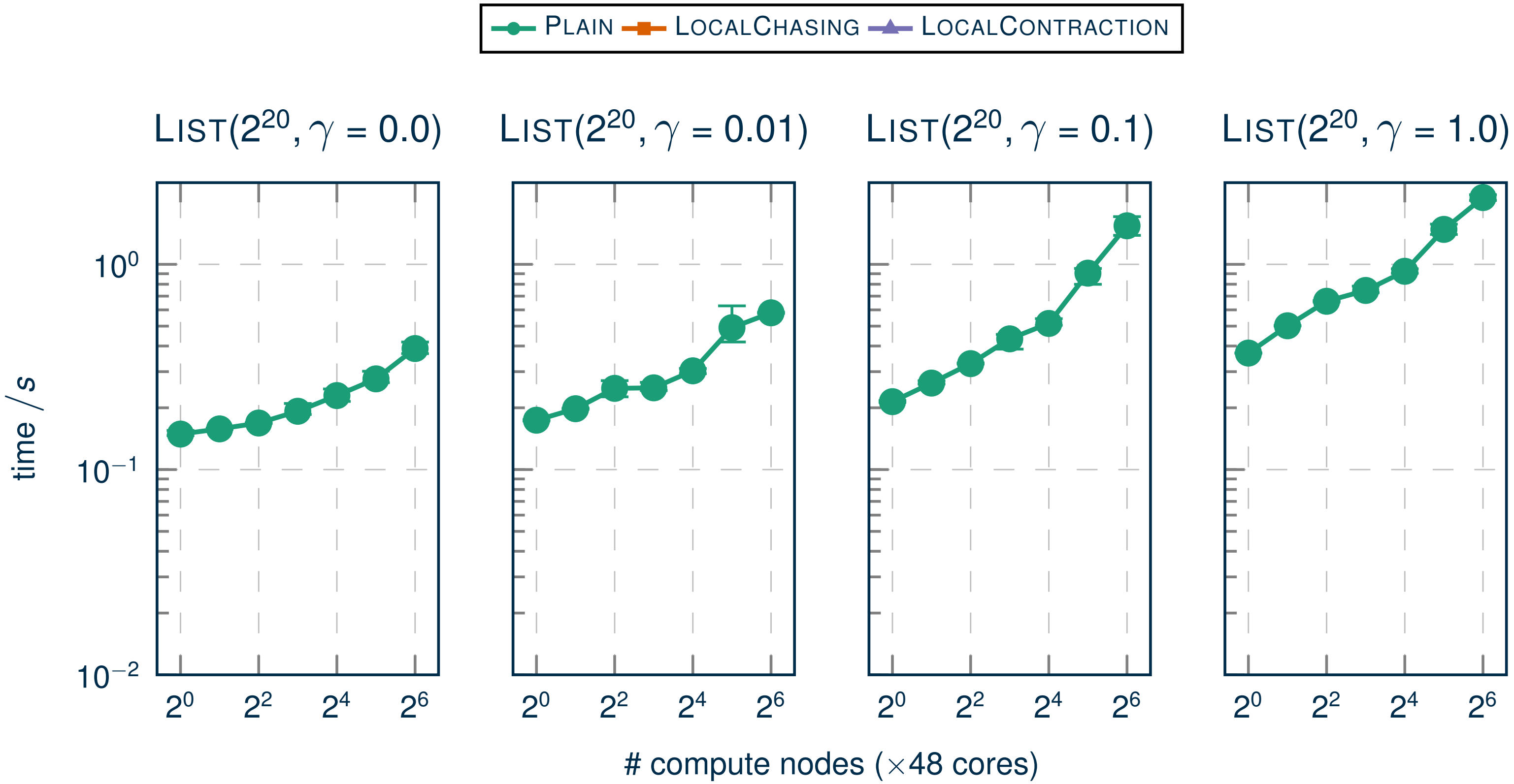




Making it Scale

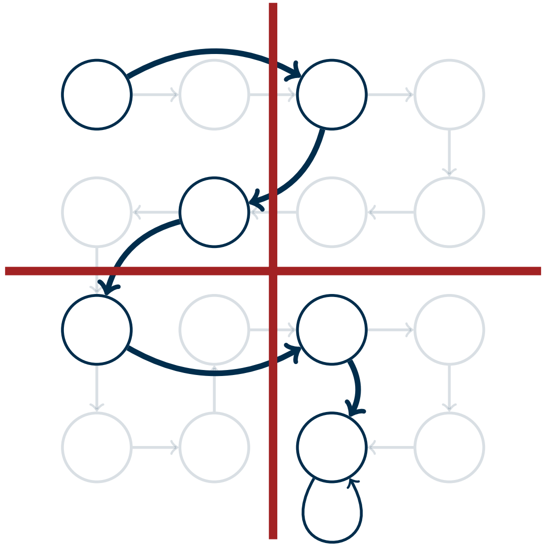
Exploiting Locality

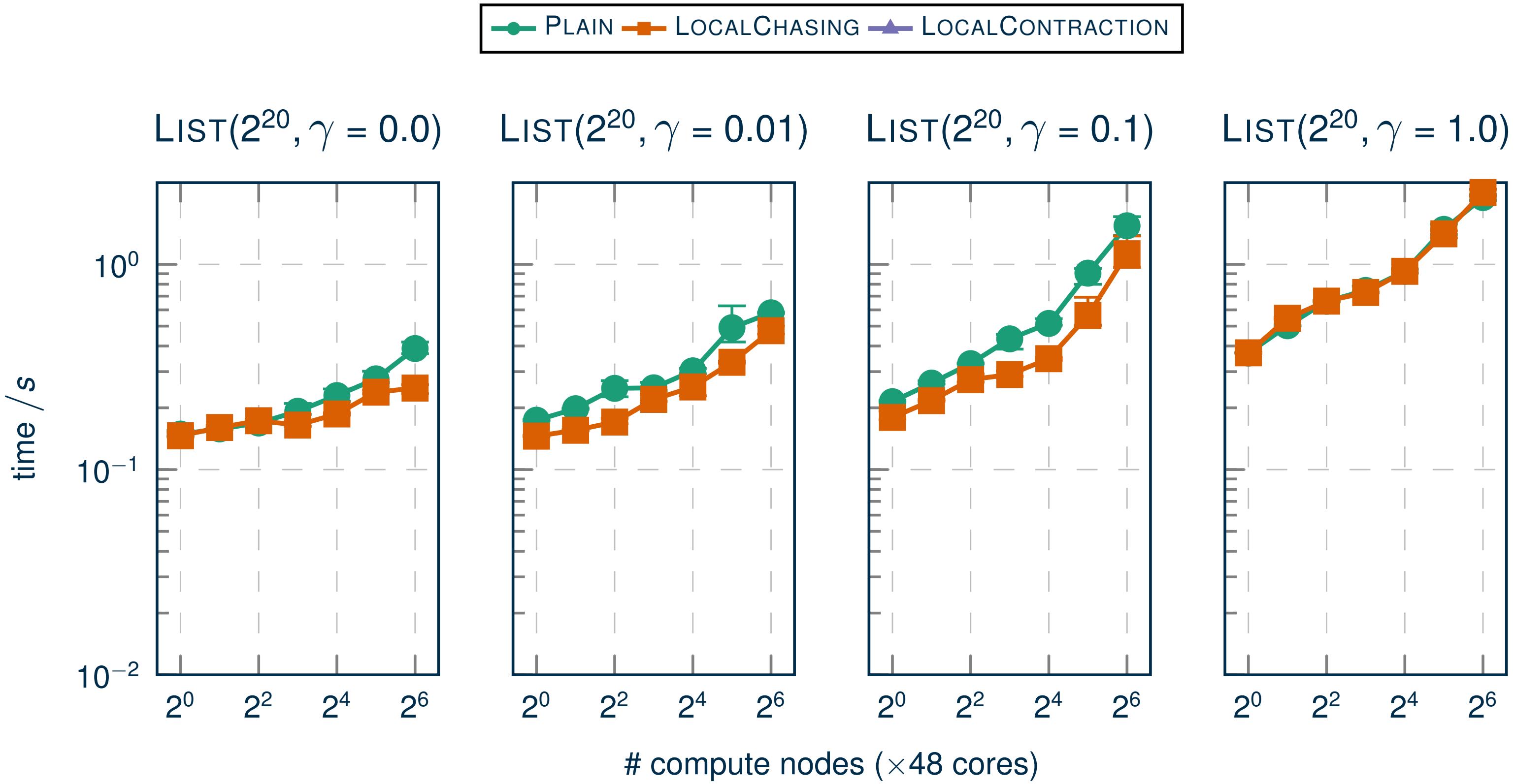




Making it Scale

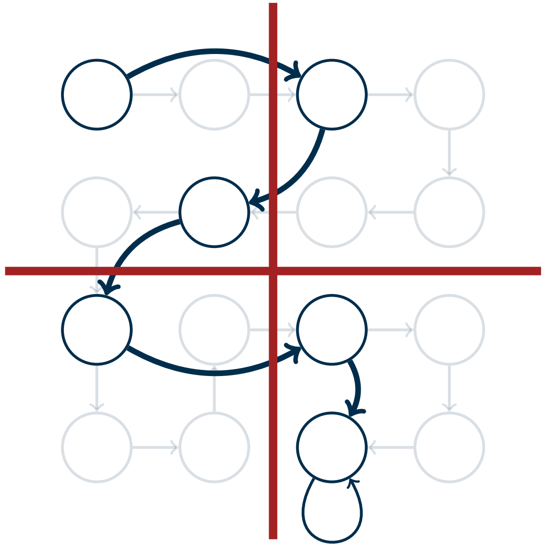
Exploiting Locality

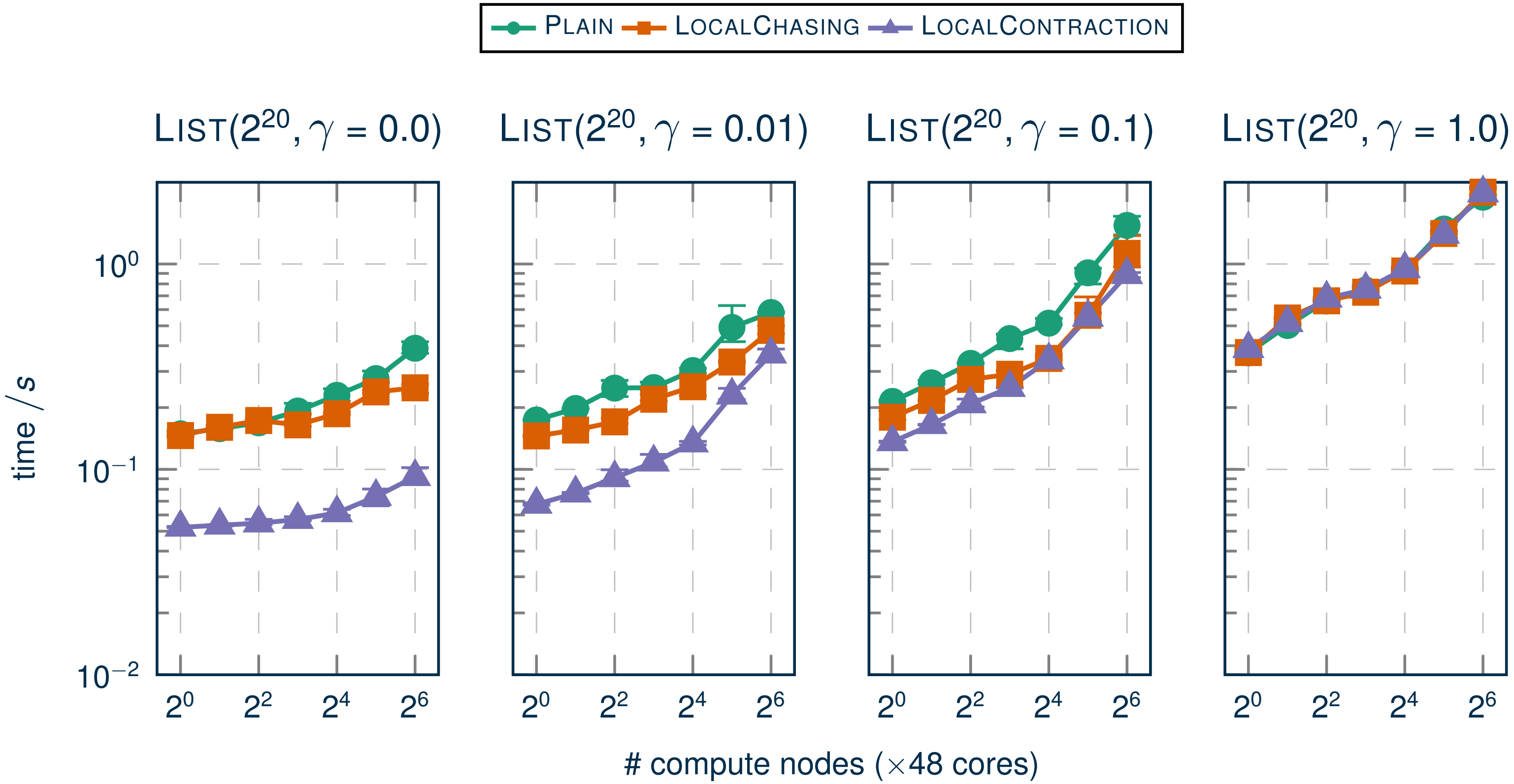




Making it Scale

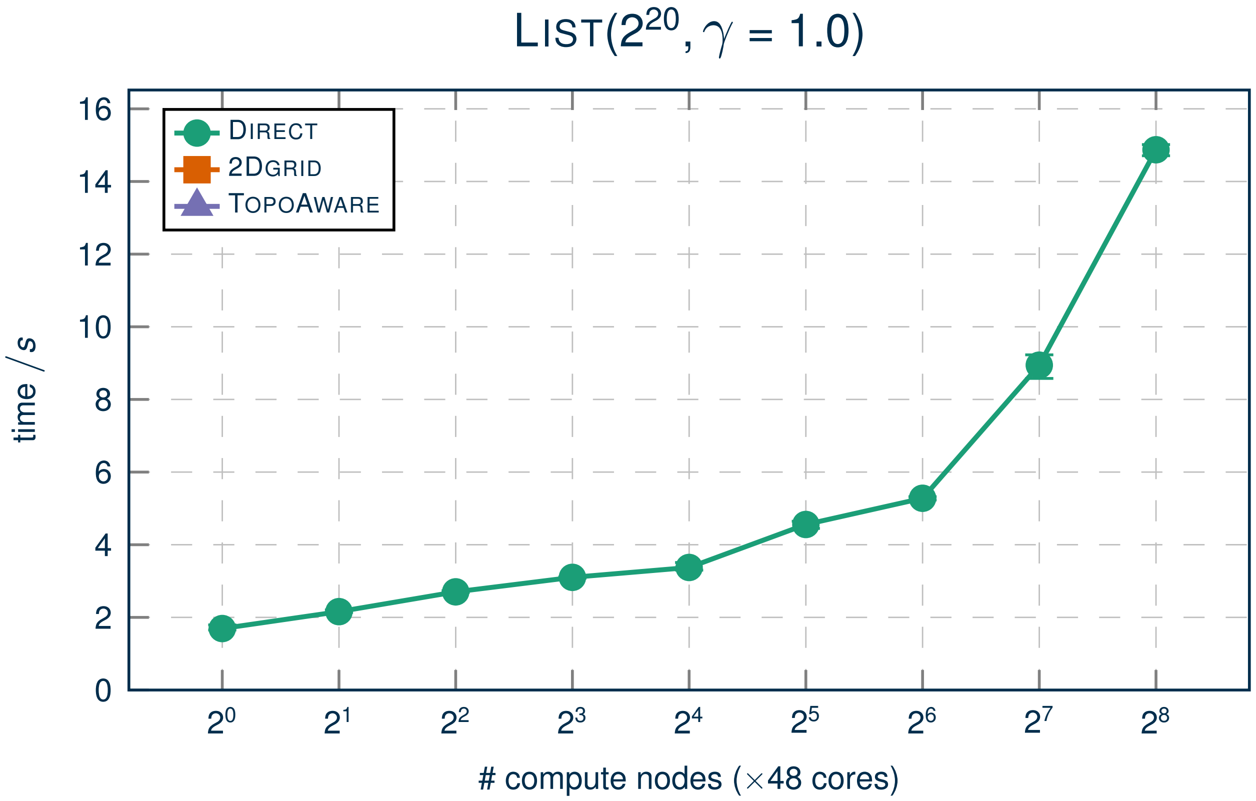
Exploiting Locality



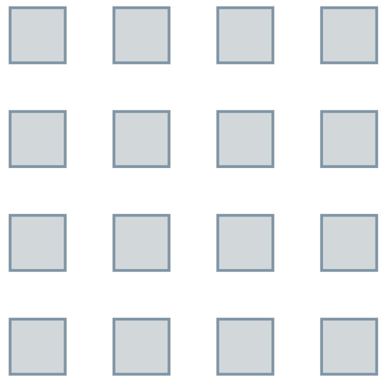


Making it Scale

Reducing Latency via Indirection

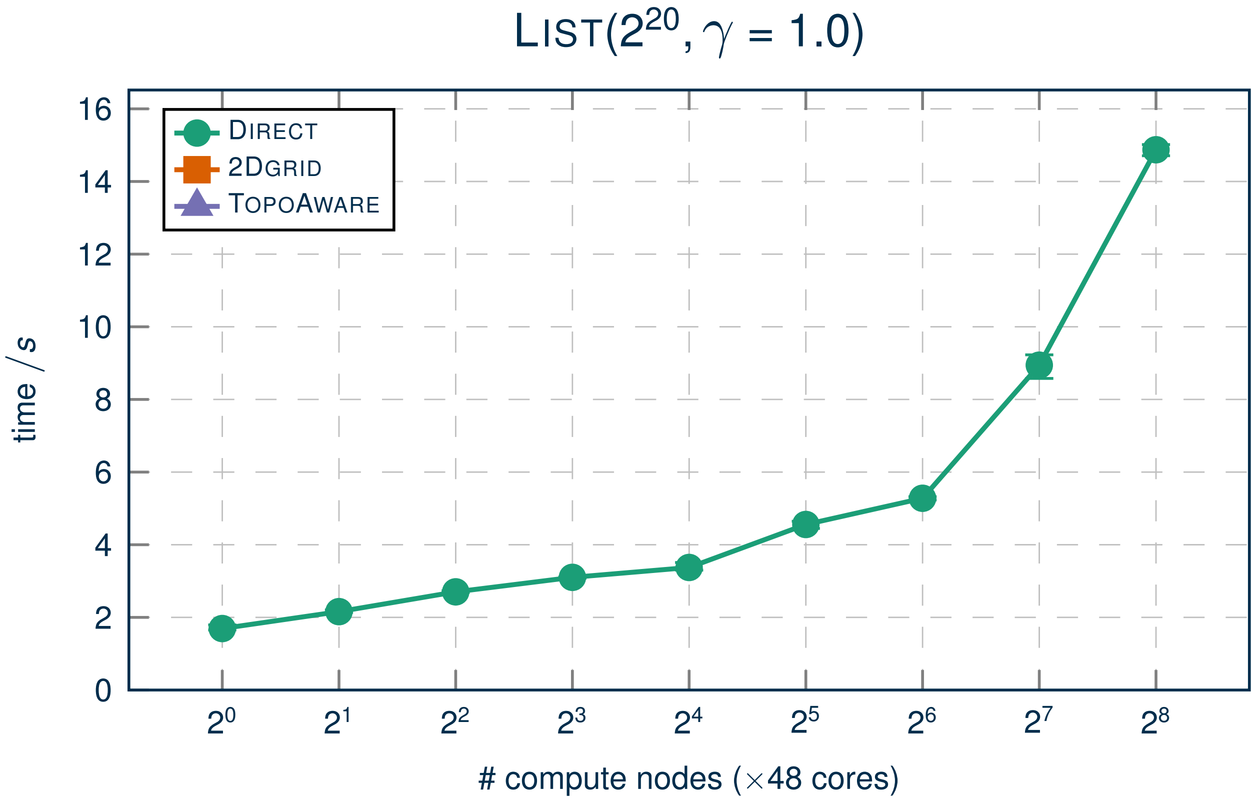


grid-based indirection

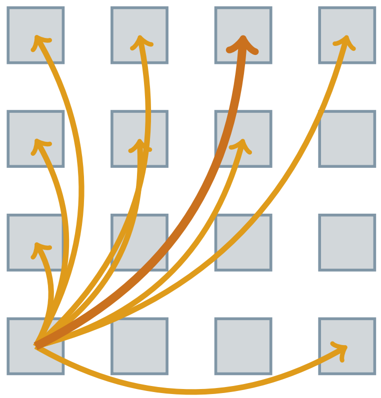


Making it Scale

Reducing Latency via Indirection

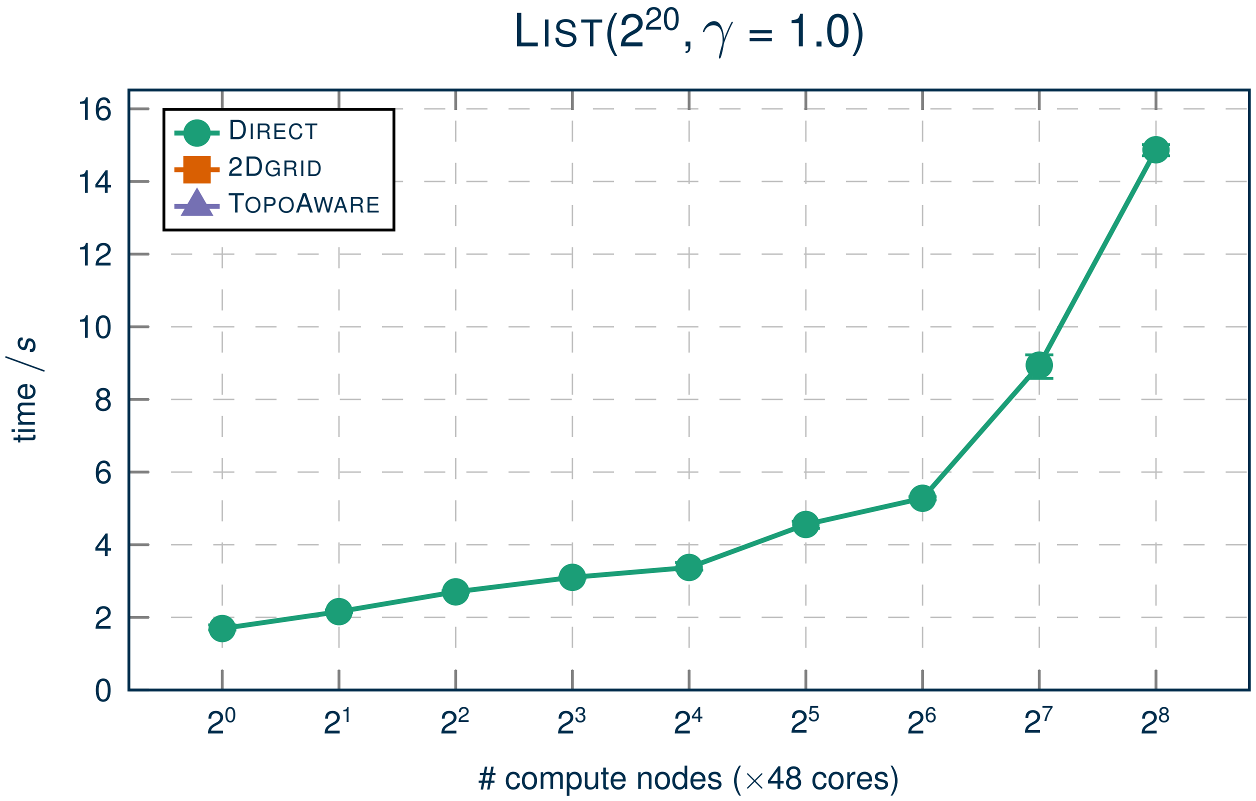


grid-based indirection

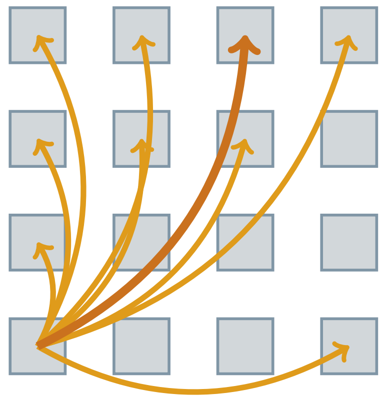


Making it Scale

Reducing Latency via Indirection



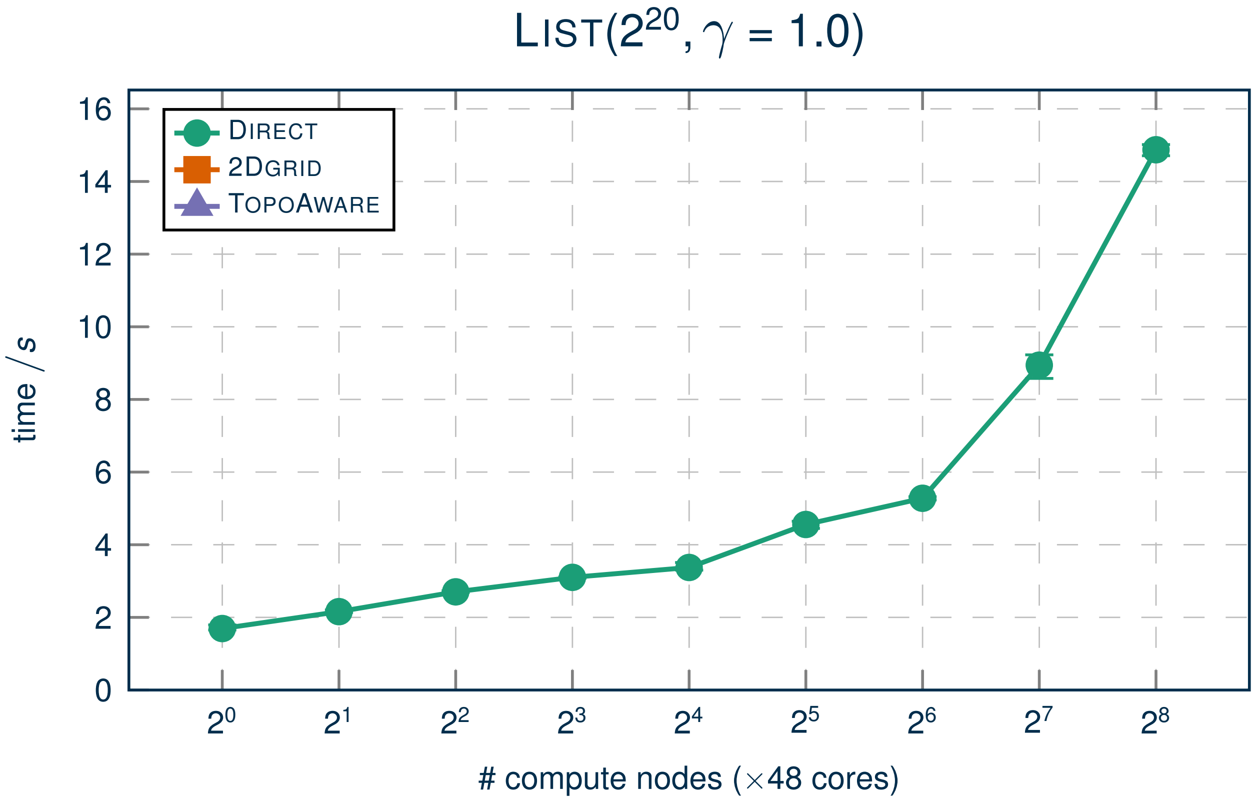
grid-based indirection



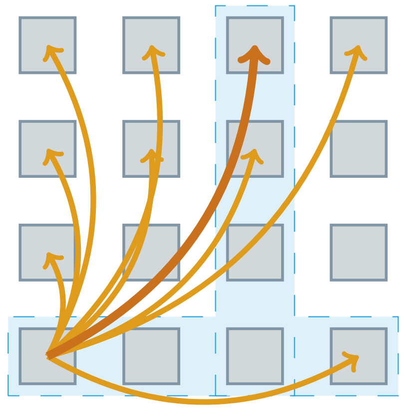
sending takes $p \quad (\alpha + \beta\ell)$

Making it Scale

Reducing Latency via Indirection



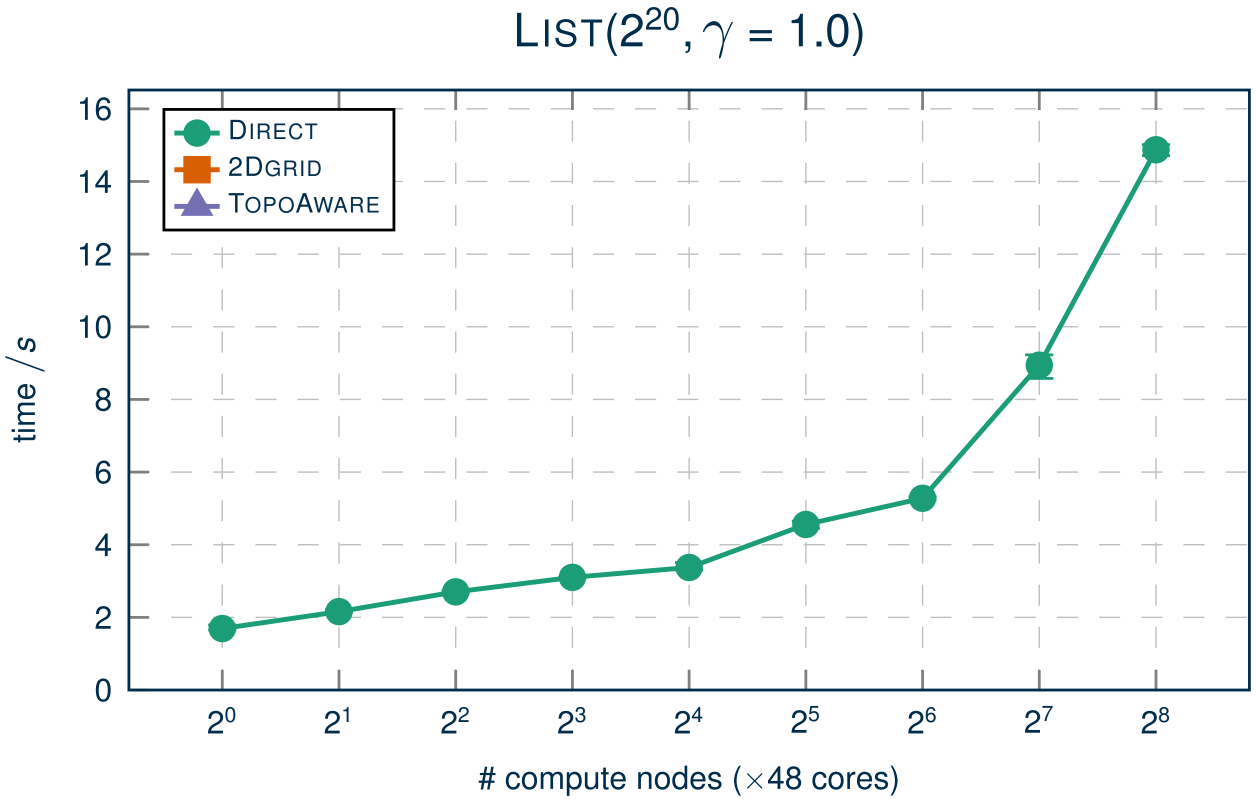
grid-based indirection



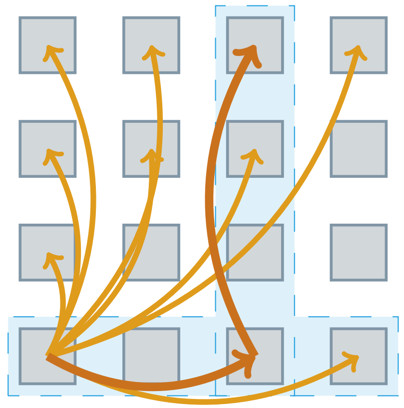
sending takes
$p \quad (\alpha + \beta\ell)$

Making it Scale

Reducing Latency via Indirection



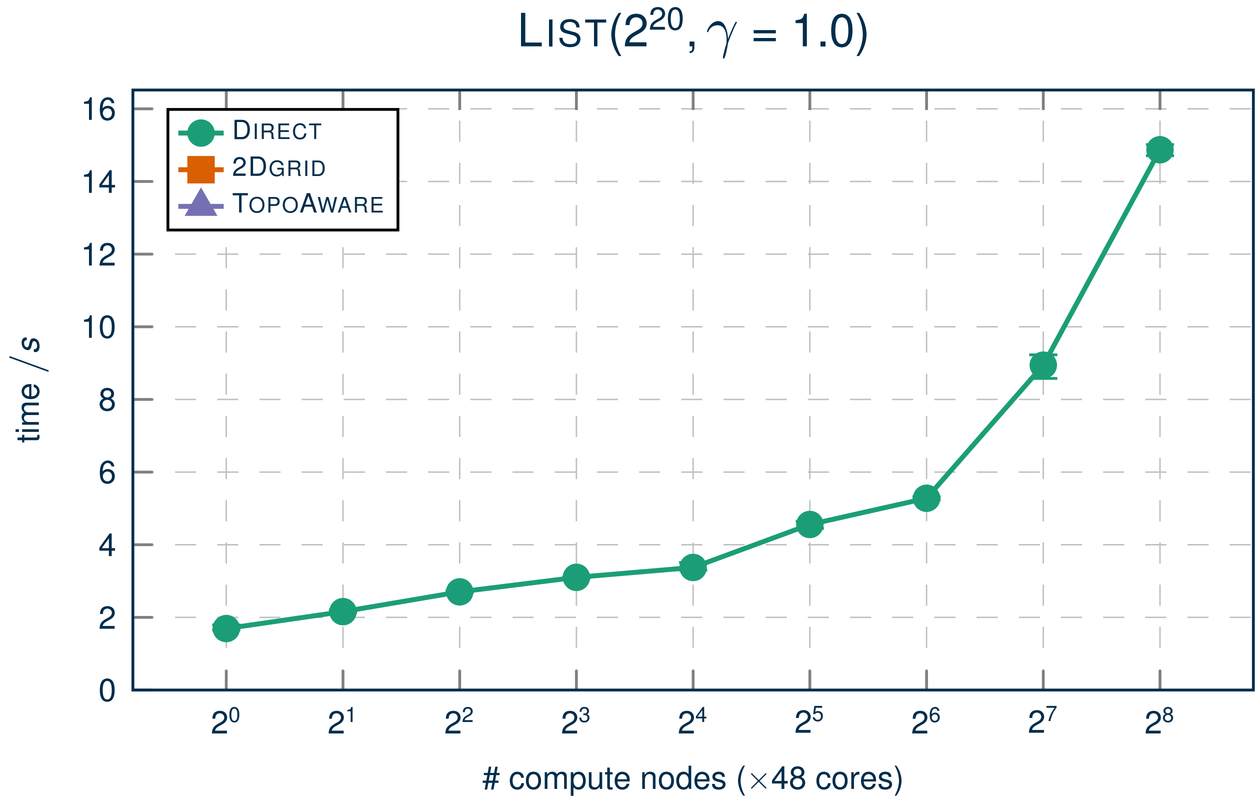
grid-based indirection



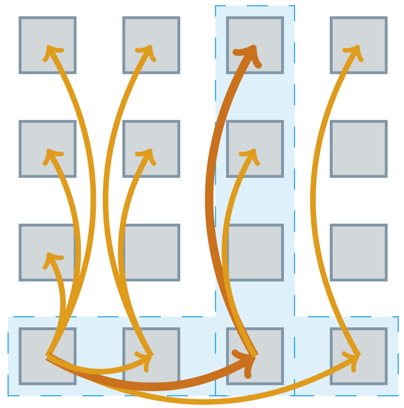
sending takes $p\ (\alpha + \beta\ell)$

Making it Scale

Reducing Latency via Indirection



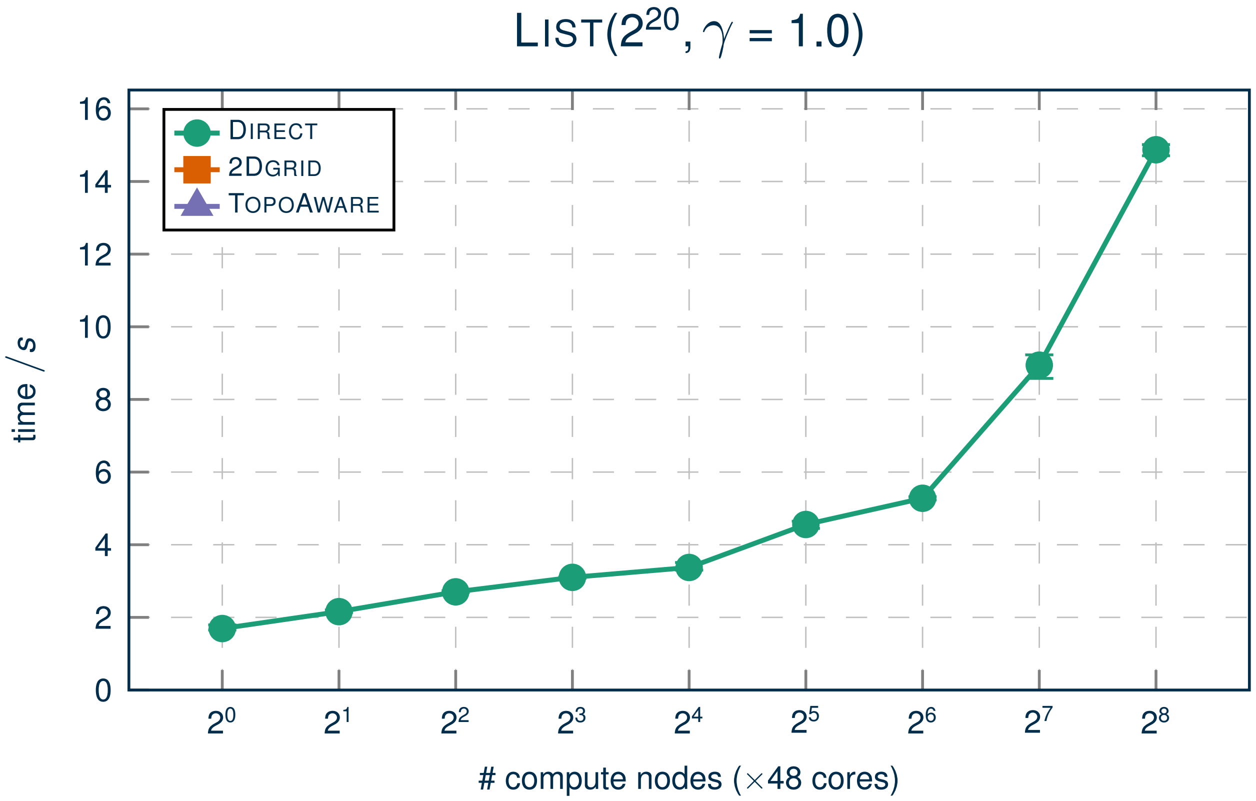
grid-based indirection



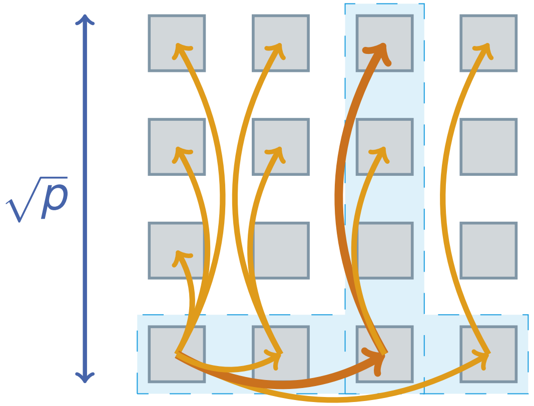
sending takes $p\ (\alpha + \beta\ell)$

Making it Scale

Reducing Latency via Indirection



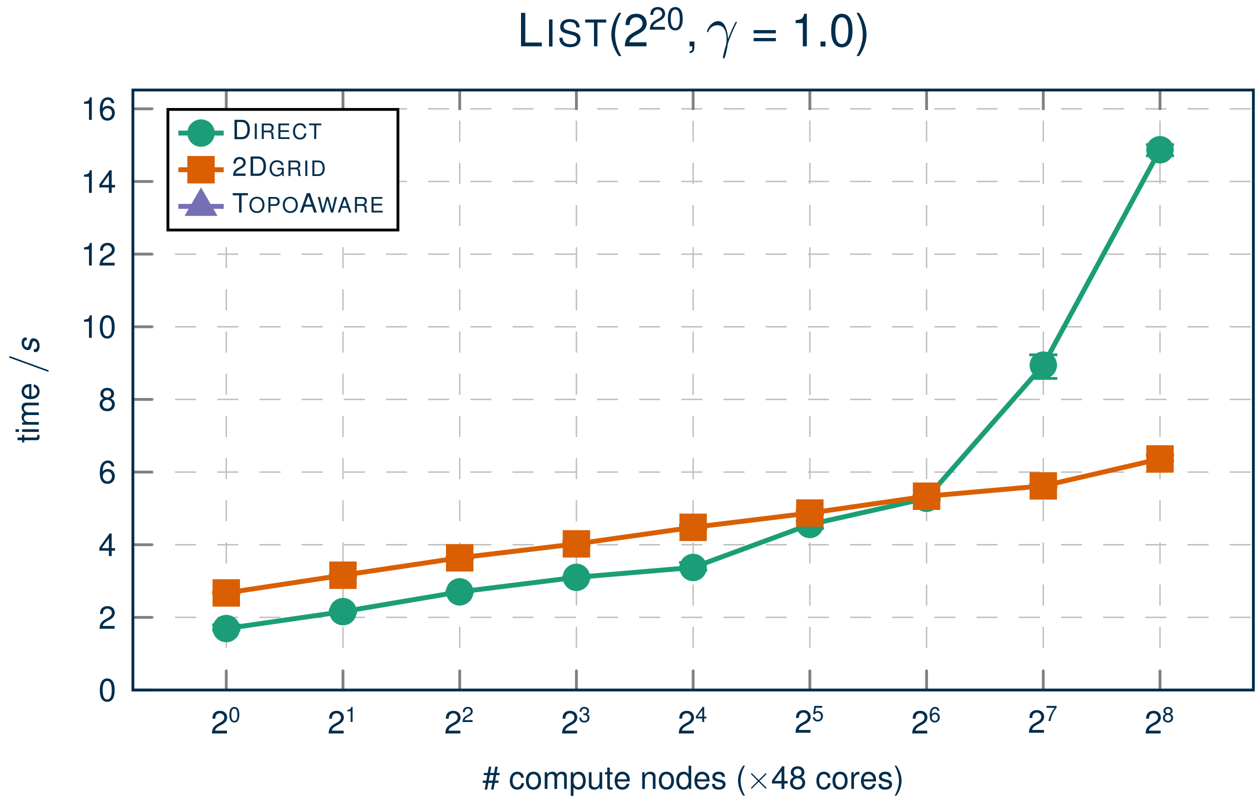
grid-based indirection

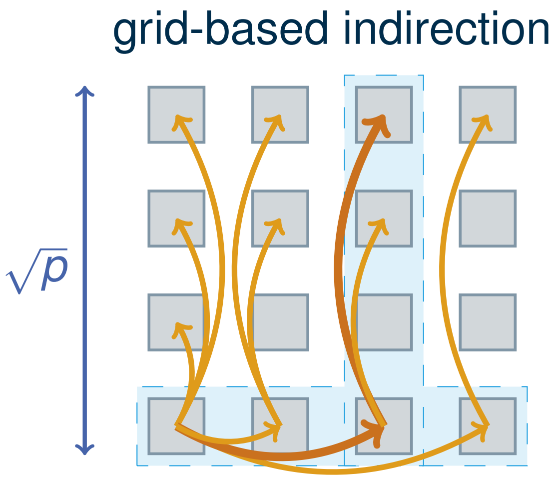


sending takes $2\sqrt{p}(\alpha + \beta\ell)$

Making it Scale

Reducing Latency via Indirection

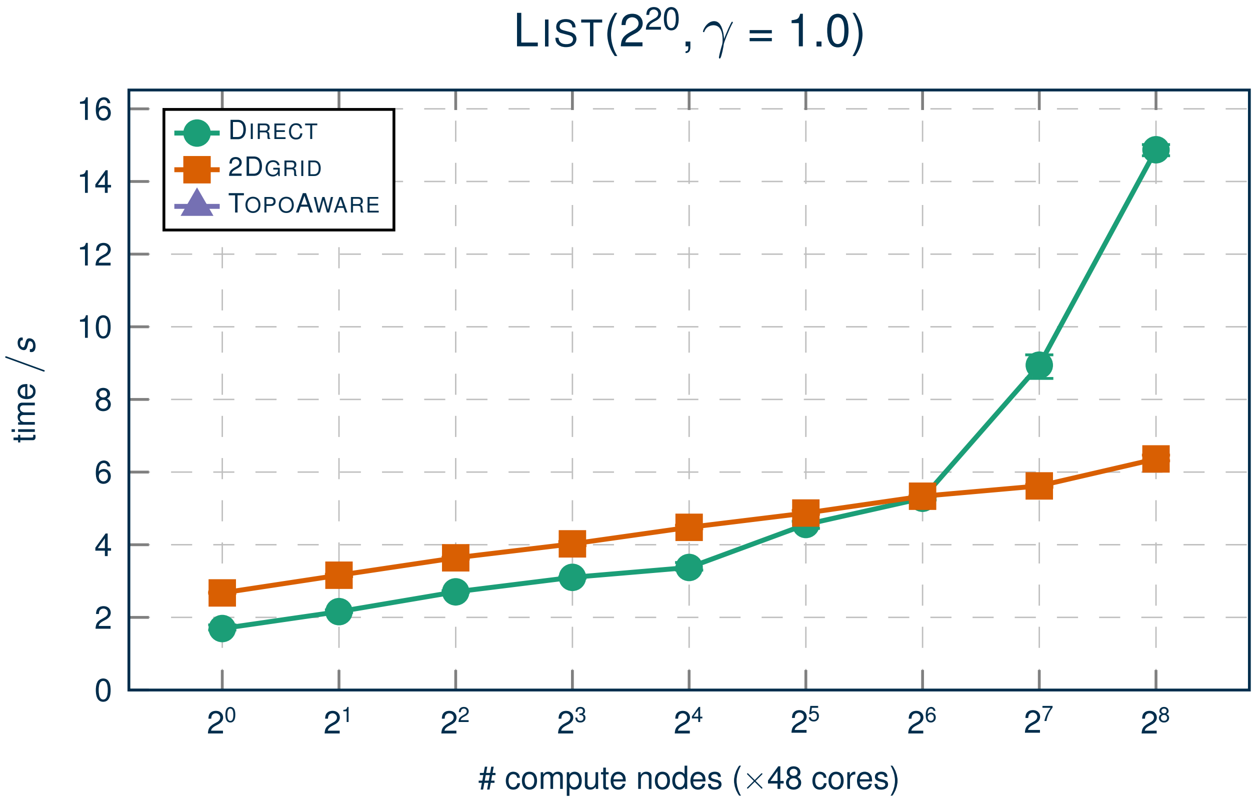




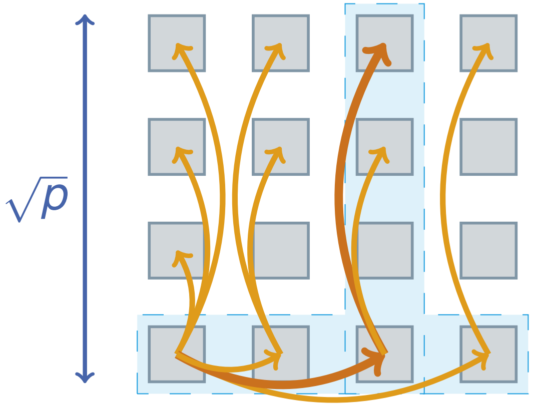
sending takes $2\sqrt{p}(\alpha + \beta\ell)$

Making it Scale

Reducing Latency via Indirection

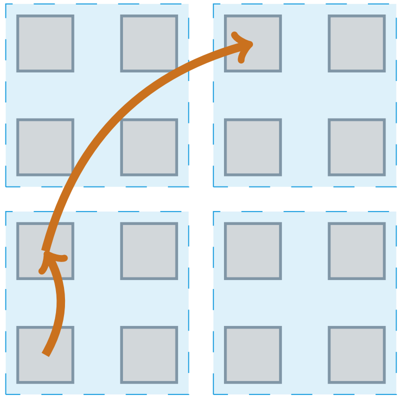


grid-based indirection



sending takes $2\sqrt{p}(\alpha + \beta\ell)$

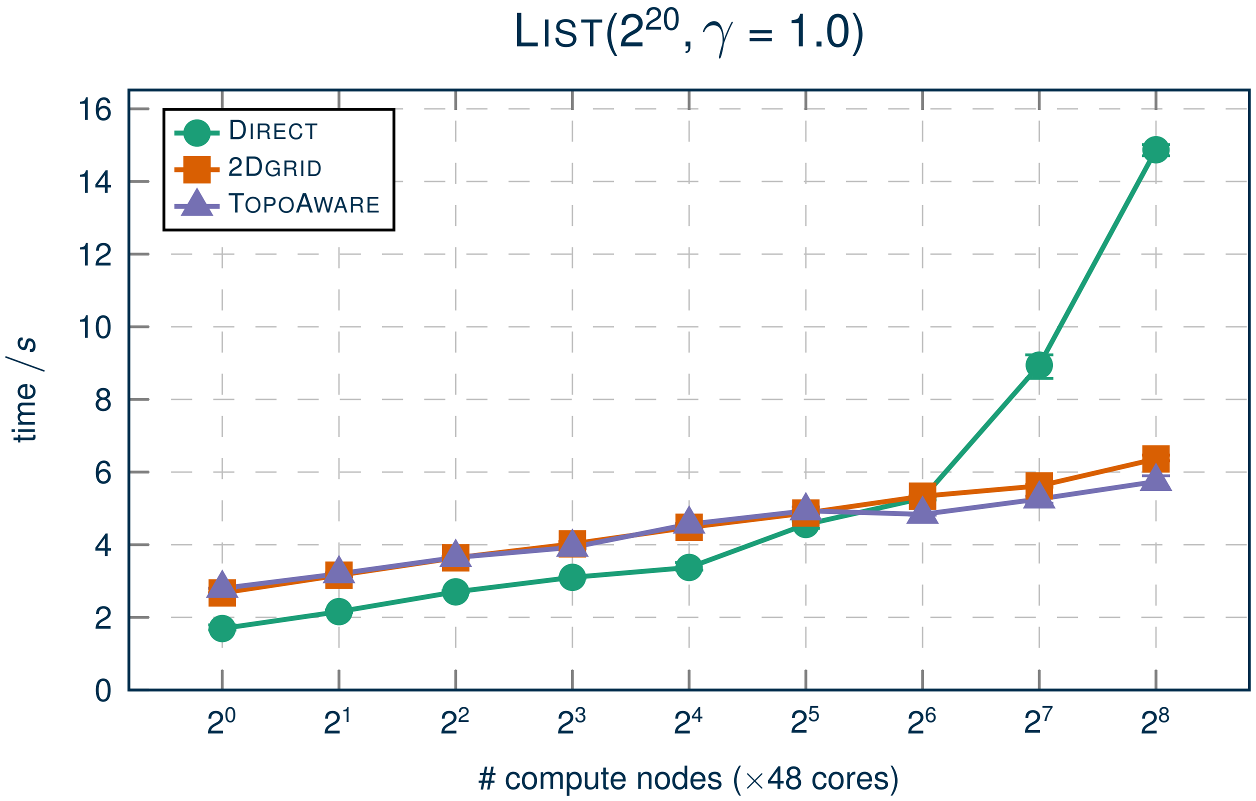
topology-aware indirection



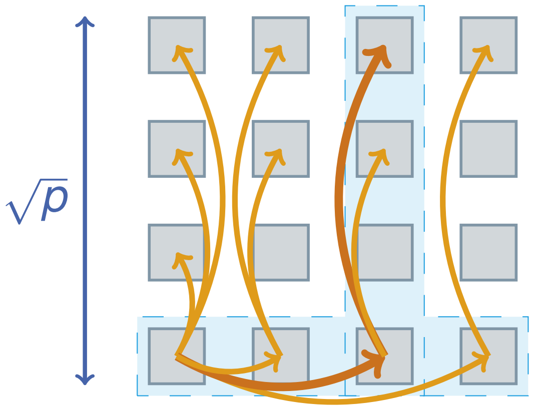
p = cp’

Making it Scale

Reducing Latency via Indirection

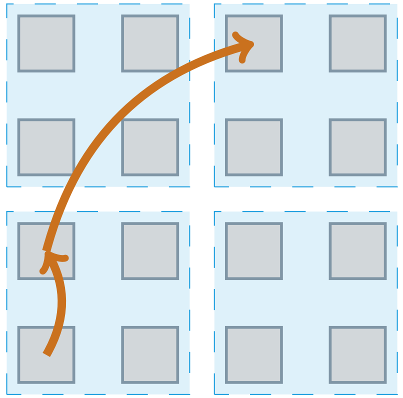


grid-based indirection



sending takes $2\sqrt{p}(\alpha + \beta\ell)$

topology-aware indirection



p = cp'

Making it Scale

Choosing the Number of Rulers

indirection levels

$$\underbrace{\alpha d p^{1/d} \frac{n}{r}}_{\text{ruler-chasing rounds}} \stackrel{!}{=} \underbrace{\beta d \frac{r \log^2 p}{p}}_{\text{base-case subproblem}}$$

Making it Scale

Choosing the Number of Rulers

indirection levels

$$\underbrace{\alpha d p^{1/d} \frac{n}{r}}_{\text{ruler-chasing rounds}} \stackrel{!}{=} \underbrace{\beta d \frac{r \log^2 p}{p}}_{\text{base-case subproblem}}$$

Observation

optimal number of rulers

$$r^* = \Theta\left(\frac{\sqrt{\alpha n p^{1+1/d}/\beta}}{\log p}\right)$$

yields running time

$$T^*(n,p) = \mathcal{O}\left(\frac{d\beta n}{p} + \alpha d p^{1/d} \log p + d\sqrt{\frac{\alpha\beta n}{p^{1-1/d}}} \log p\right)$$

Making it Scale

Choosing the Number of Rulers

indirection levels

$$\underbrace{\alpha d p^{1/d} \frac{n}{r}}_{\text{ruler-chasing rounds}} \stackrel{!}{=} \underbrace{\beta d \frac{r \log^2 p}{p}}_{\text{base-case subproblem}}$$

Observation

optimal number of rulers

$$r^* = \Theta\left(\frac{\sqrt{\alpha n p^{1+1/d} / \beta}}{\log p}\right)$$

yields running time

$$T^*(n,p) = \mathcal{O}\left(\frac{d\beta n}{p} + \alpha d p^{1/d} \log p + d\sqrt{\frac{\alpha\beta n}{p^{1-1/d}}} \log p\right)$$

- in practice: $r = c\sqrt{np^{1+1/d}} / \log p$, tuned $c = 2.0$ experimentally

Making it Scale

Choosing the Number of Rulers

indirection levels

$$\underbrace{\alpha d p^{1/d}\frac{n}{r}}_{\text{ruler-chasing rounds}} \stackrel{!}{=} \underbrace{\beta d \frac{r \log^2 p}{p}}_{\text{base-case subproblem}}$$

Observation

optimal number of rulers

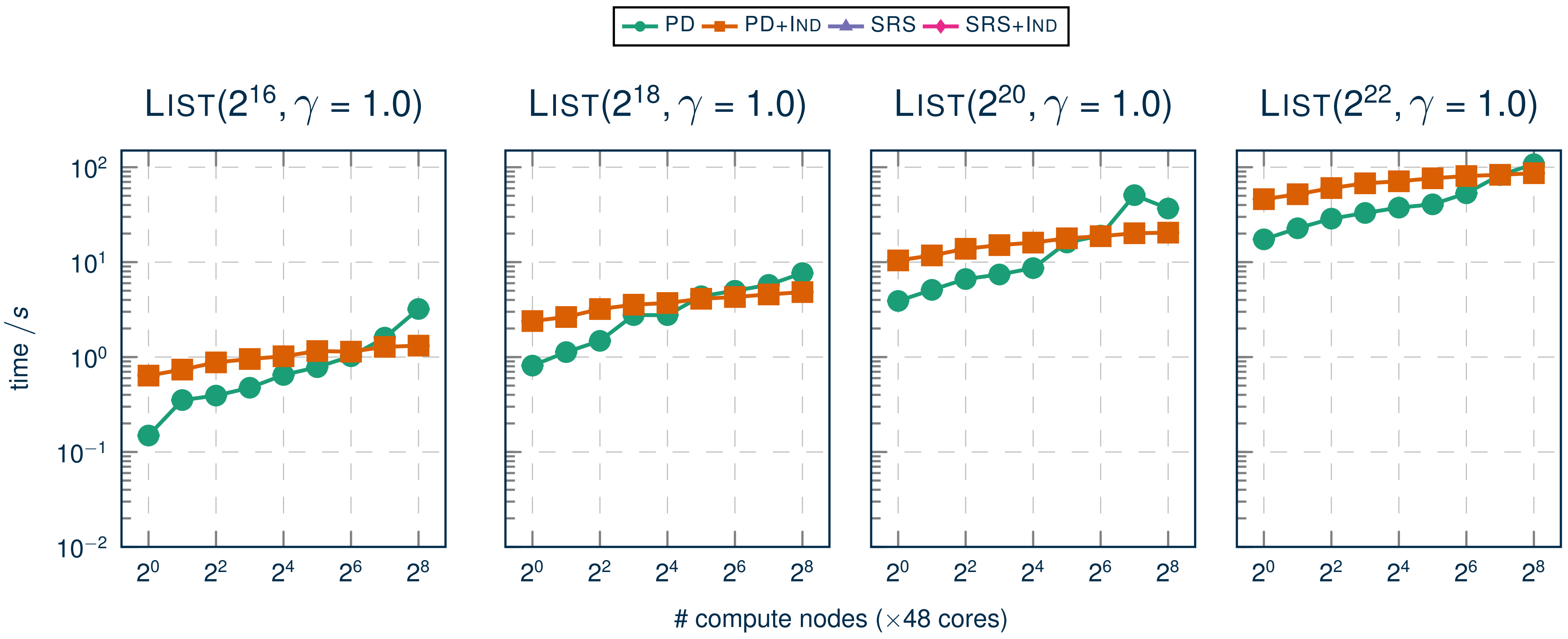
$$r^* = \Theta\left(\frac{\sqrt{\alpha n p^{1+1/d}/\beta}}{\log p}\right)$$

yields running time

$$T^*(n,p) = \mathcal{O}\left(\frac{d\beta n}{p} + \alpha d p^{1/d}\log p + d\sqrt{\frac{\alpha\beta n}{p^{1-1/d}}}\log p\right)$$

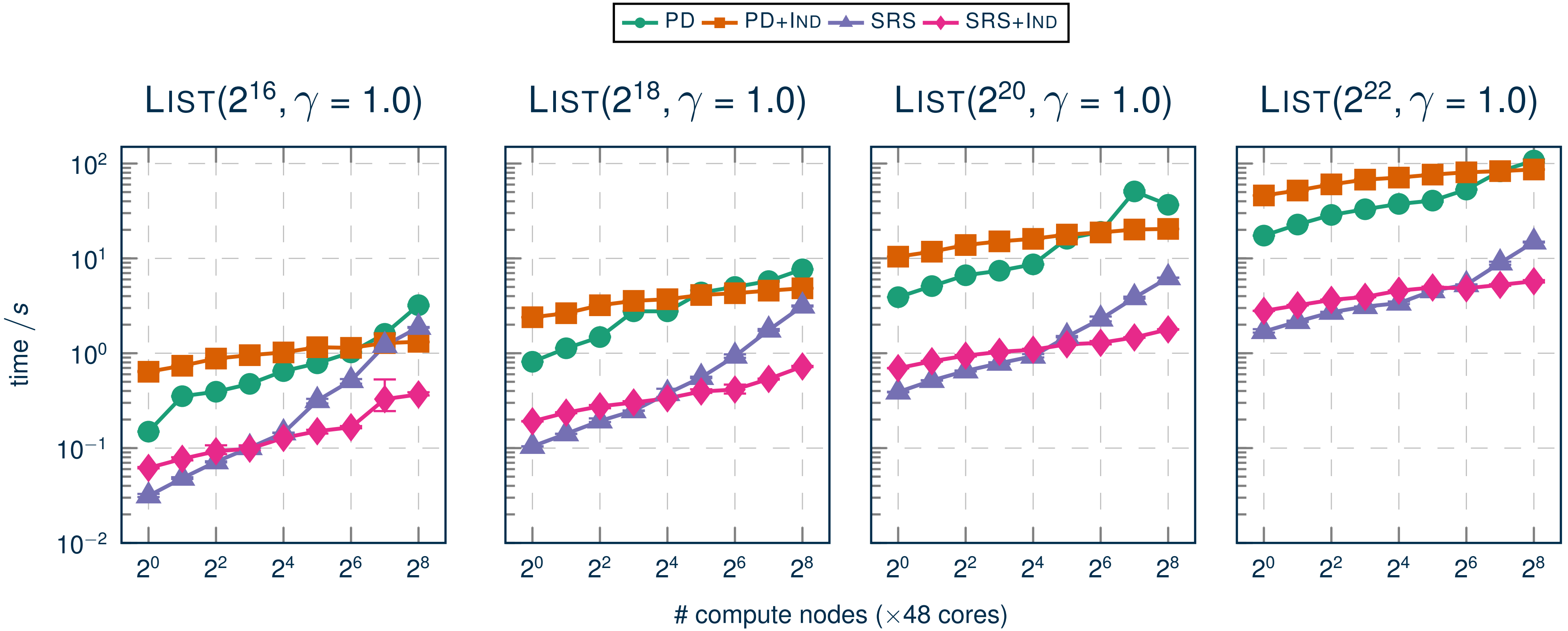
- in practice: $r = c\sqrt{np^{1+1/d}}/\log p$, tuned $c = 2.0$ experimentally
- **corollary:** efficient whenever $n/p \gg \frac{\alpha}{\beta}p^{1/d}\log^2 p$ elements per PE

Scaling Experiments



- SRS+IND scales to 12 288 cores, up to 15× faster than pointer doubling
- scaled SRS+IND to 24 576 cores, ranking over 100 billion elements

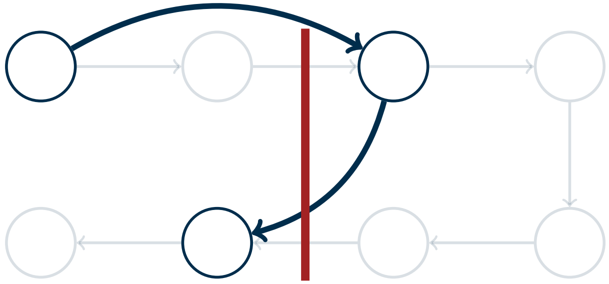
Scaling Experiments



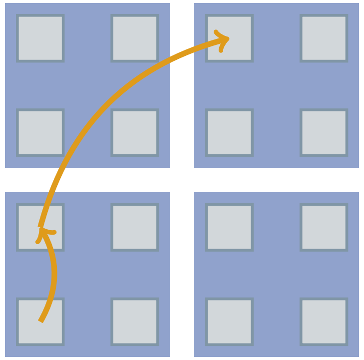
- SRS+IND scales to 12 288 cores, up to 15× faster than pointer doubling
- scaled SRS+IND to 24 576 cores, ranking over 100 billion elements

Engineering Scalable List Ranking

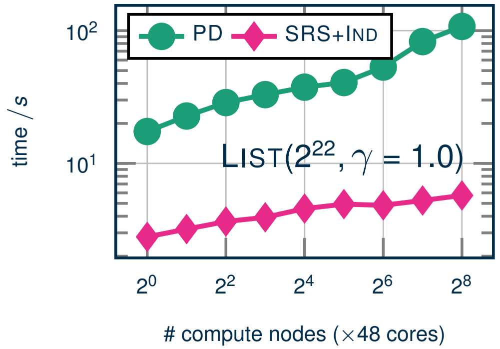
Conclusion



Exploit Locality



Indirect Communication



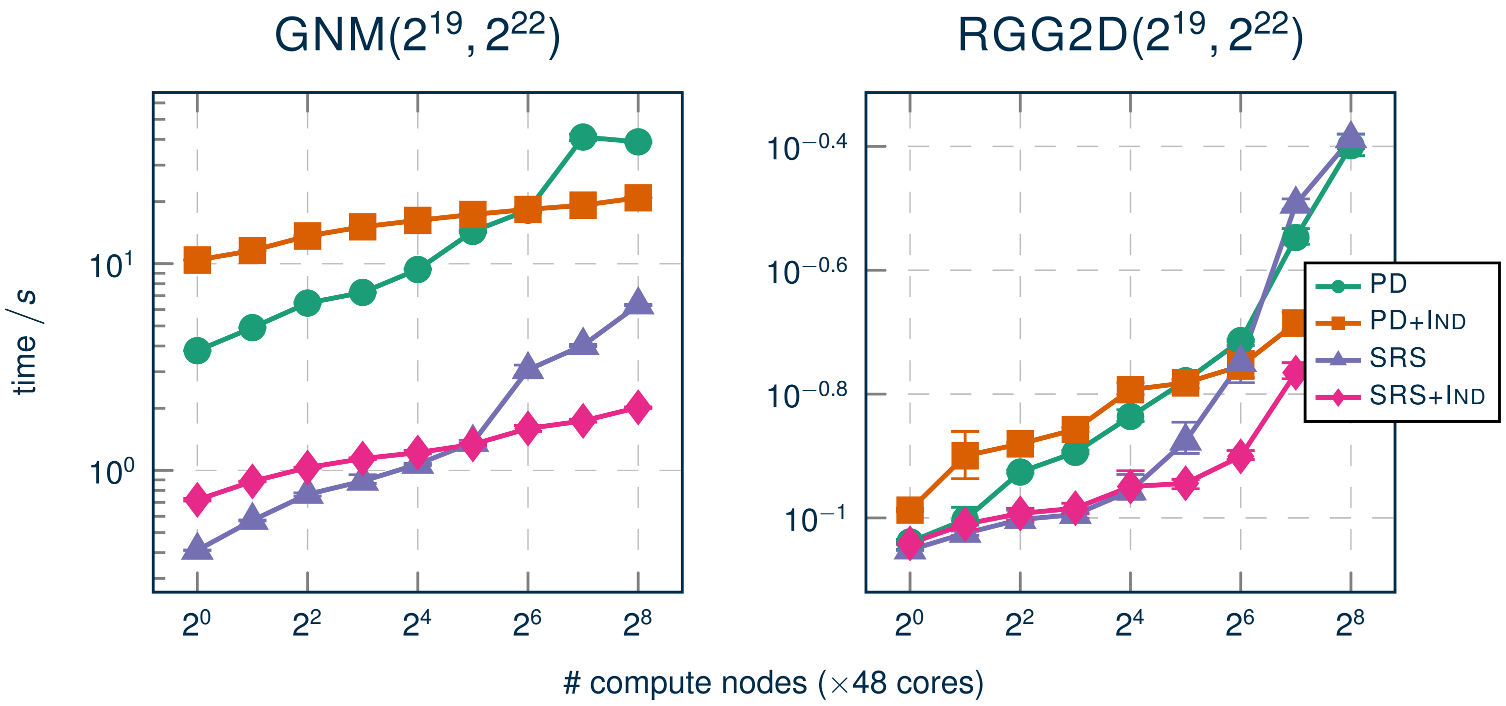
Scales to 24 576 Cores

- list ranking is entirely **communication-bound** $\rightarrow$ prime candidate for studying scalable irregular algorithms
- first **practical evaluation** of sparse ruling-set with spawning + scalability analysis
- two orders of magnitude larger scale than previous work

Appendix

More Scaling Experiments

Euler-Tour Instances from Random Graphs



- RGG2D has high locality: ≈ 1% of elements remain after local preprocessing, so both +IND variants are latency-bound
- GNM has almost no locality and behaves like random lists

Time Breakdown for Indirection Variants