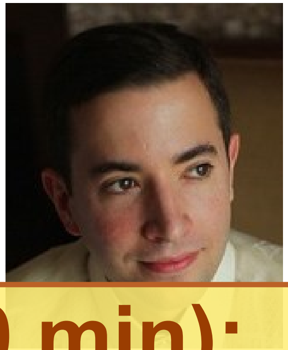
Optimal Parallel Algorithms in the Binary-Forking Model

A joint work with

Yan Gu, UC Riverside

SPAA 2020 (virtual)

July 15, 2020



Guy Blelloch, Jeremy Fineman, Yihan Sun

CMU Georgetown UC Riverside

Full version of this talk (40 min):

https://www.cs.ucr.edu/~ygu/video/bf.mp4

(or just search “Yan Gu UCR”)

1 The binary-forking model

Don't we have lots of computational models already? Why more?

2 Optimal algorithms in the new model

Sorting/semisorting, list/tree contraction, set operations (union, intersect, difference), random permutation, RMQ, etc.

Are they useful and inspiring?

Development of parallel hardware (from 2005)

2020 — **Intel® Xeon® Platinum 8260**

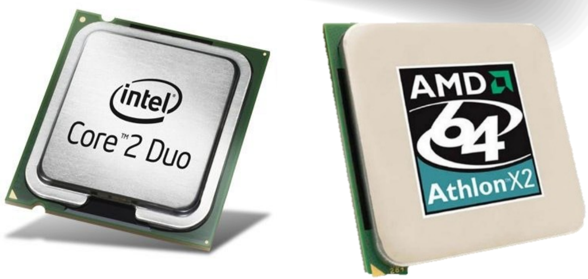
1 of up to 4 sockets:
28 cores
56 threads
38.5 MB LLC
Up to 4.5 TB memory

2017 — **Intel® Xeon Phi™**

72 cores
(288 threads)

For computation intensive applications

2005 — **Intel® Core™2 Duo (E6320) / AMD Athlon 64 X2**



Parallel sorting

[GBSS'15, OKFS'19, AWFS'17, CBB+'15, ABPS'15, BGS'10, BGHS'12, RPC+'11, SBF+'12, MSL+'15 ......]

In-memory databases

AerospikeDBS, Ancelus, Apache Ignite, ArangoDB, Hyper, Kinetica, MemSQL, SQLite, IMDB, SQL Server, Monet DB, SAP HANA, VoltDB, ...

Parallel data structure

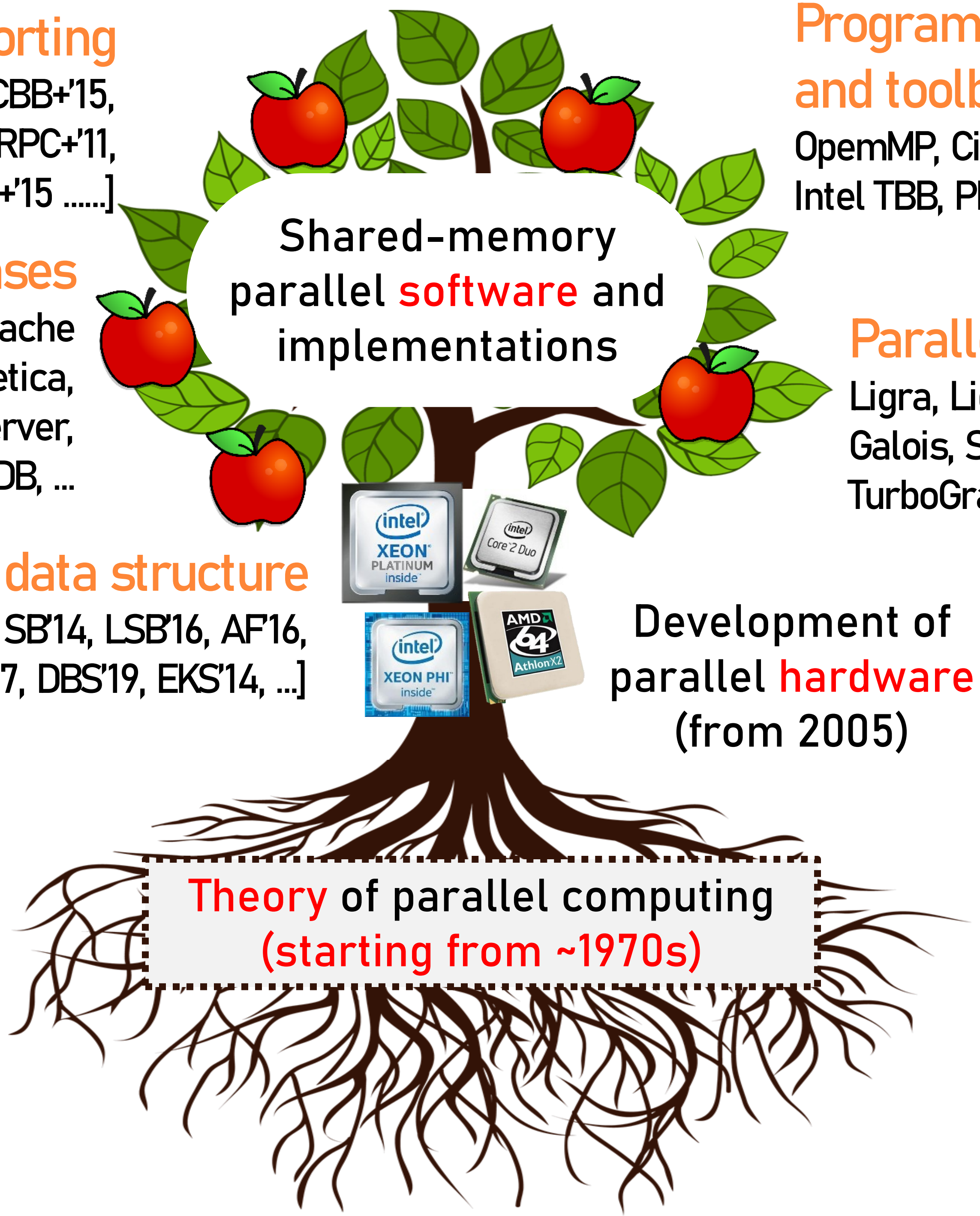
[BFGS'12, BFS'16, SB'14, LSB'16, AF'16, SB'19, YRP'07, BFF+'07, DBS'19, EKS'14, ...]

Programming languages and toolboxes

OpemMP, Cilk, Java fork-join, X10, Intel TBB, PBBS, MCSTL, ...

Parallel graph processing

Ligra, Ligra+, GraphIt, Julienne, Aspen, GBBS, Galois, STINGER, PRISM, X-Stream, Ringo, TurboGraph, FlashGraph, Kaskade, Sage, ...



Parallel sorting

[GBSS'15, OKFS'19, AWFS'17, CBB+'15, ABPS'15, BGS'10, BGHS'12, RPC+'11, SBF+'12, MSL+'15 ......]

In-memory databases

AerospikeDBS, Ancelus, Apache Ignite, ArangoDB, Hyper, Kinetica, MemSQL, SQLite, IMDB, SQL Server, Monet DB, SAP HANA, VoltDB, ...

Parallel data structure

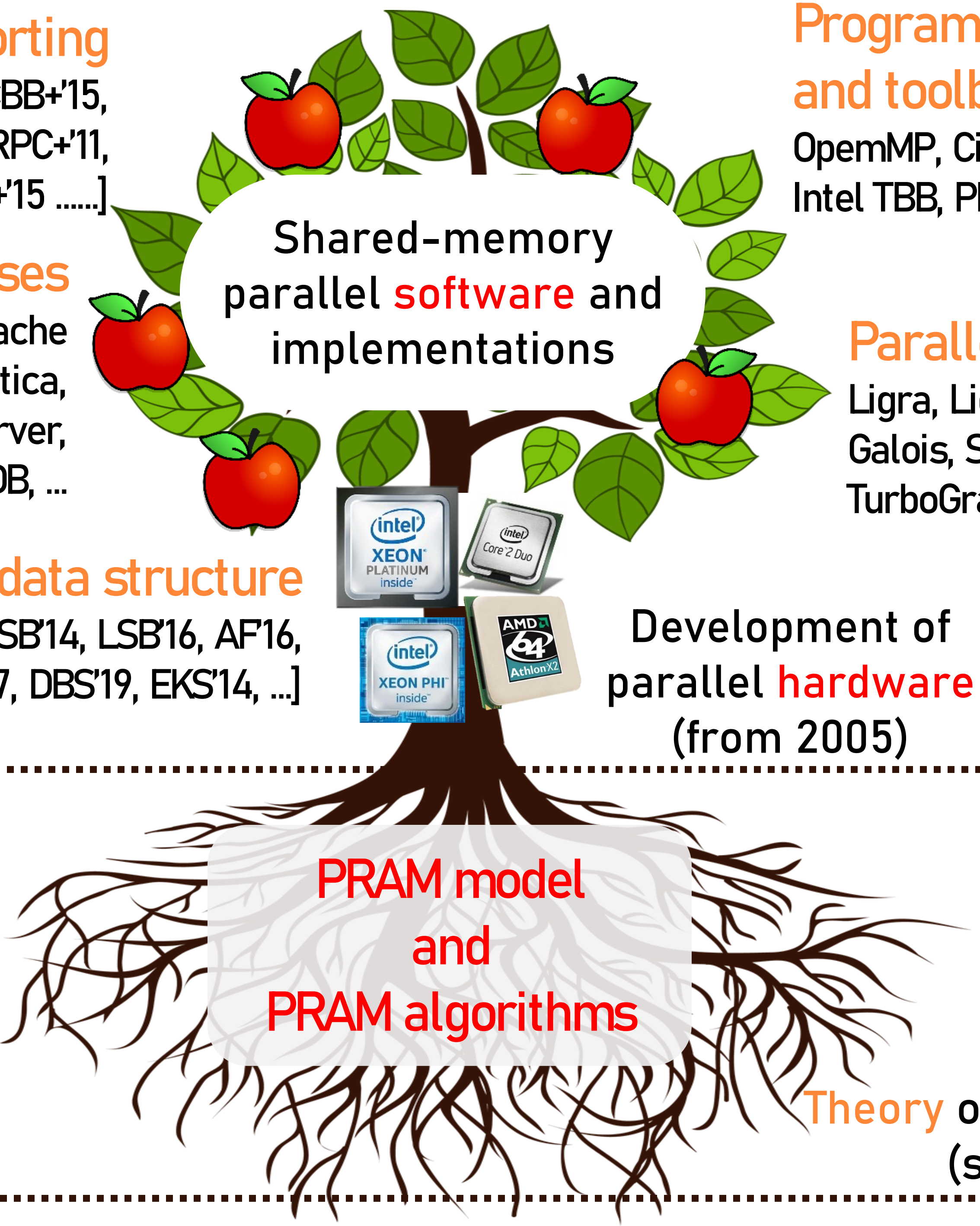
[BFGS'12, BFS'16, SB'14, LSB'16, AF'16, SB'19, YRP'07, BFF+'07, DBS'19, EKS'14, ...]

Programming languages and toolboxes

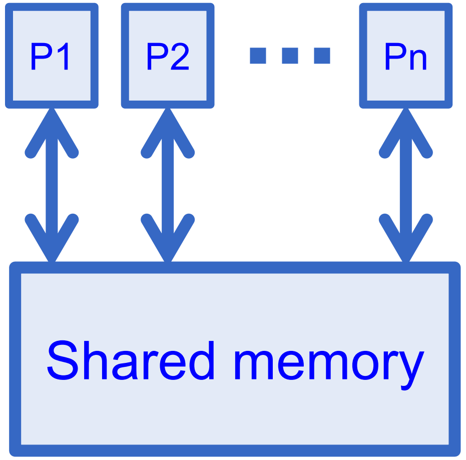
OpemMP, Cilk, Java fork-join, X10, Intel TBB, PBBS, MCSTL, ...

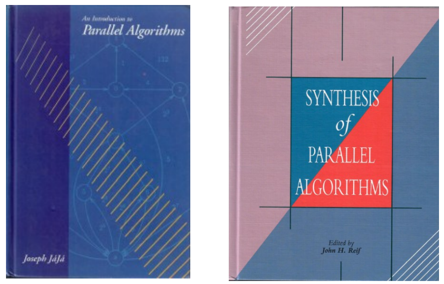
Parallel graph processing

Ligra, Ligra+, GraphIt, Julienne, Aspen, GBBS, Galois, STINGER, PRISM, X-Stream, Ringo, TurboGraph, FlashGraph, Kaskade, Sage, ...



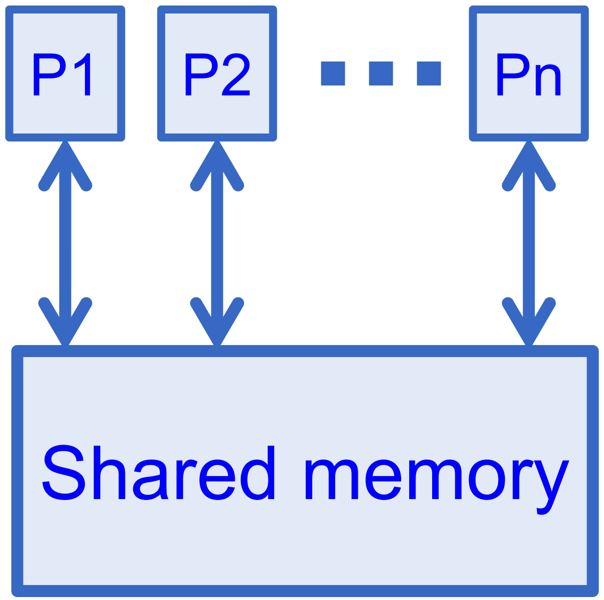
Development of parallel hardware (from 2005)





Theory of parallel computing (starting from ~1970s)

Is PRAM still the best model for designing and analyzing algorithms today?

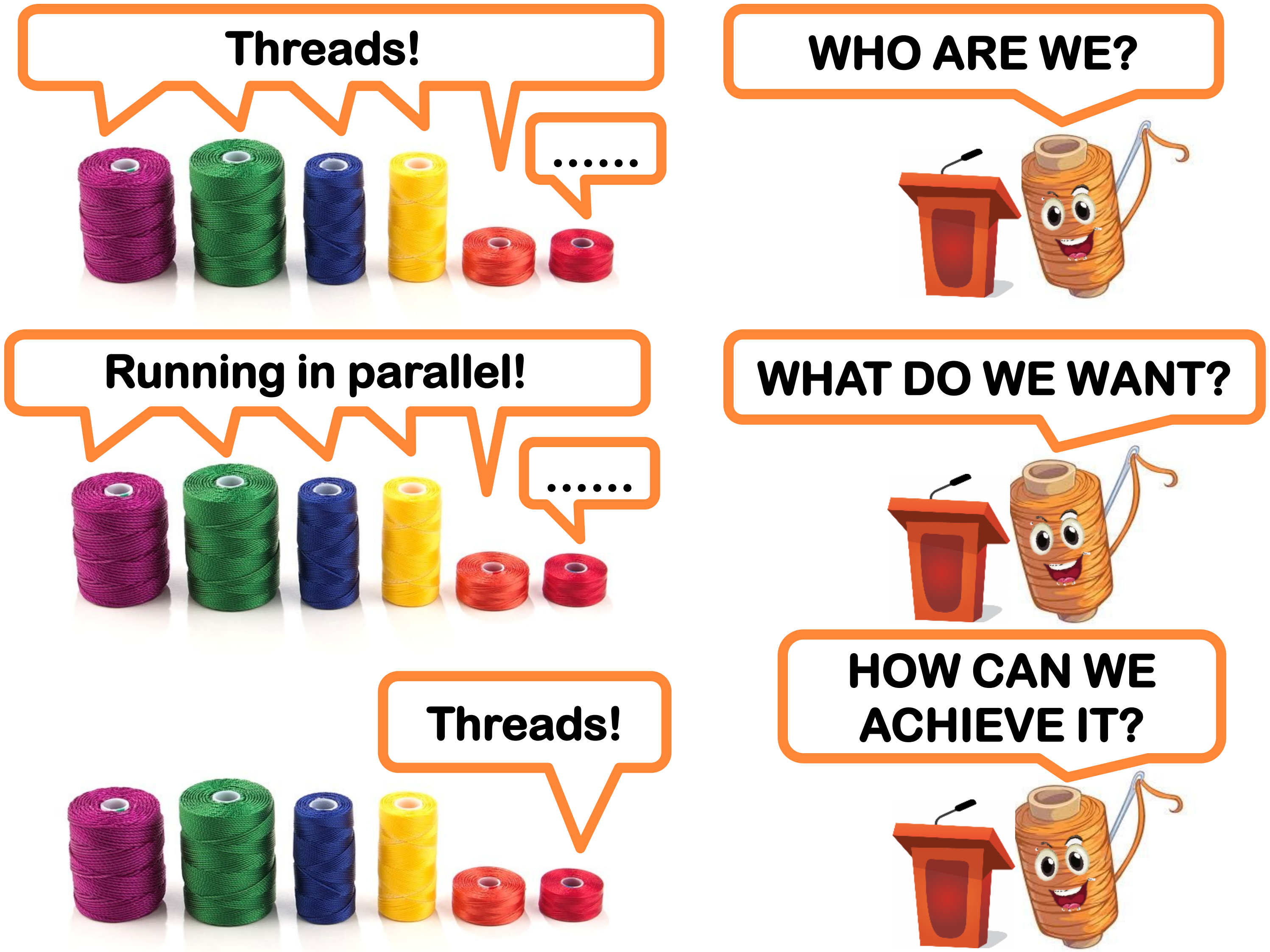


PRAM
(~1980)

Processors run in **lockstep** on a PRAM (ARBITRARY-CRCW, PRIORITY-CRCW, CREW, EREW, CRQW, etc.)

Processors are **highly synchronized**

However, modern architecture is highly asynchronous

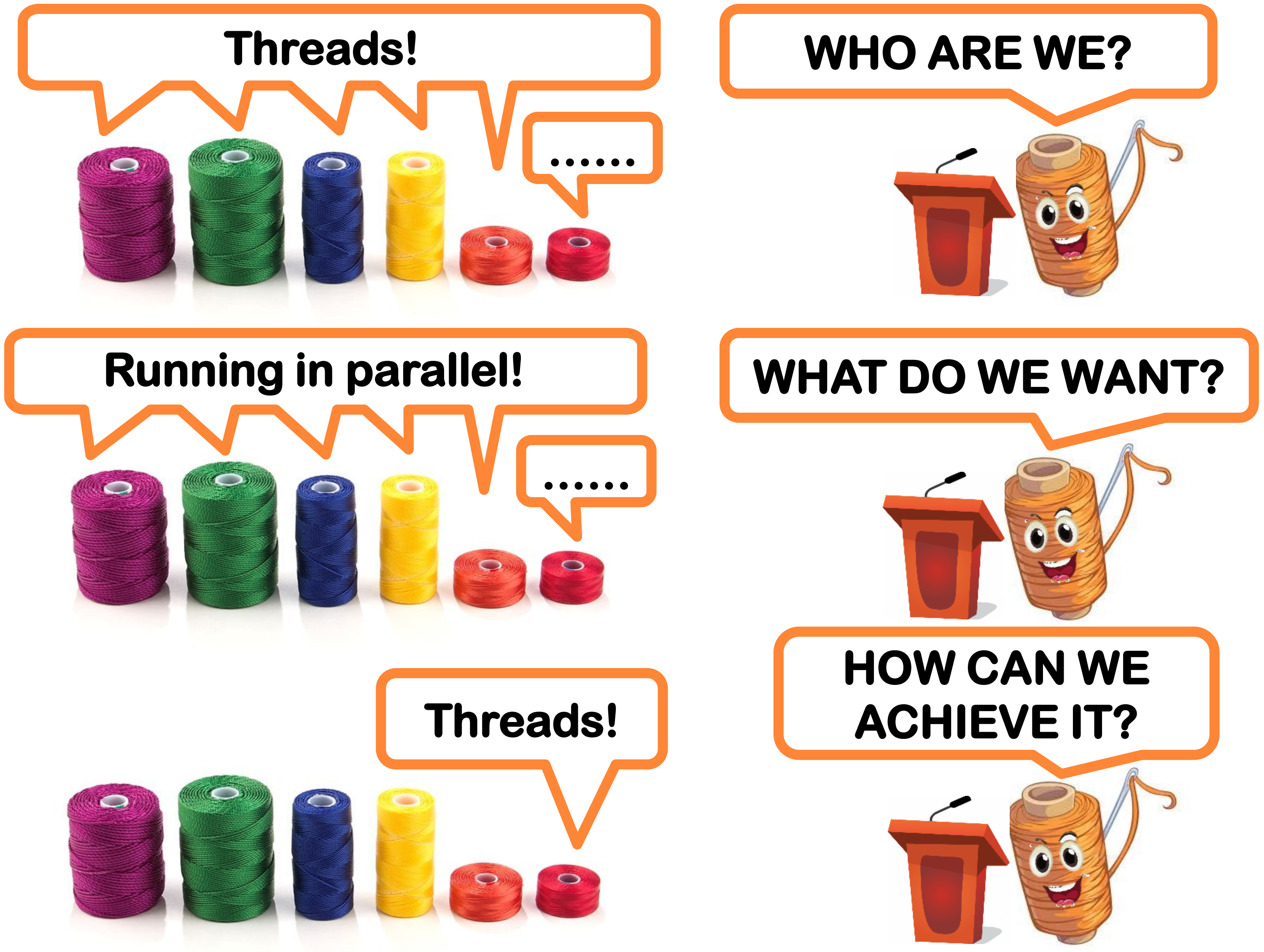


As you can see, threads are highly asynchronous.

Real architecture: processor rates vary significantly due to

- cache effects
- processor pipelines
- ILP
- vectorization
- branch prediction
- hyper-threading
- overclocking
- interrupts
- ……

Global synchronization can be extremely costly



As you can see, threads are highly asynchronous.

Real architecture: processor rates vary significantly due to

- cache effects
- processor pipelines
- ILP
- vectorization
- branch prediction
- hyper-threading
- overclocking
- interrupts
- ……

Global synchronization can be **extremely costly**

- 10^9 arithmetic operations in total
- Adding periodical global synchronization

About $\log n$, **0.7-1.7x** overhead

About $\log^2 n$, **5-13x** overhead

BSP model: give synchronization a higher weight

MPC model: only minimize synchronization rounds

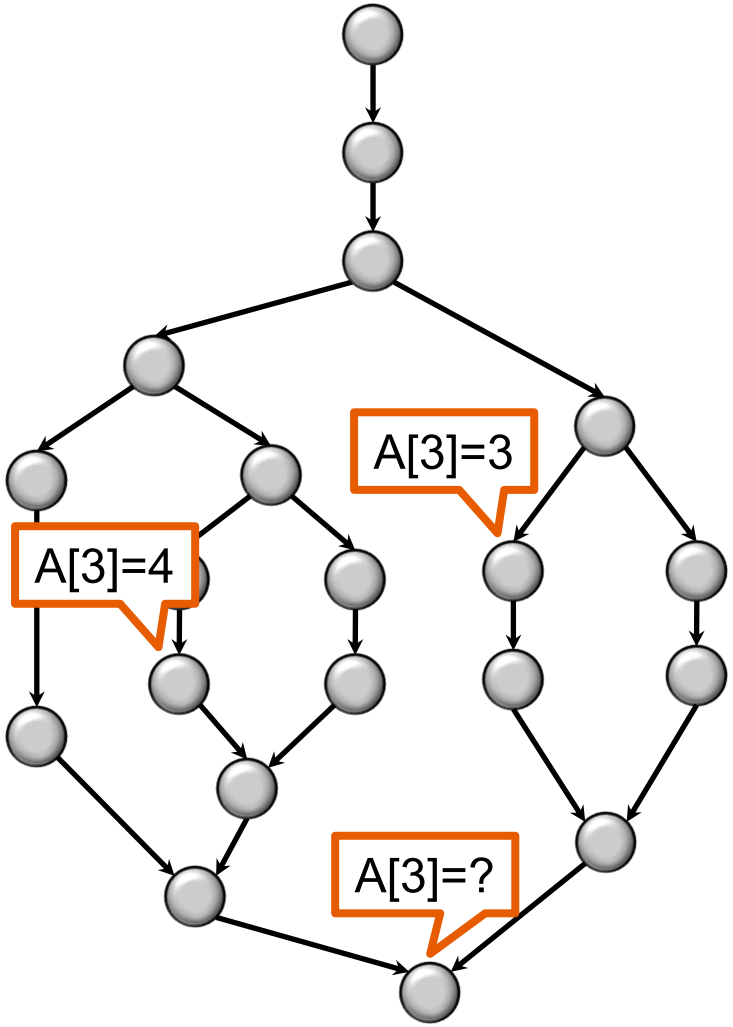
Asynchronized PRAM: synchronize only when needed

The multithreaded models [BL99, ABP01]

- A class of models
- The computational models for many recent parallel algorithms (a short list: [ABB02, AFL+14, BFGS11, BG04, BGS10, BR98, BGSS20, BST12, BL99, CGTT17, CRSB13, CR17, DST16, TYK+15])
  - Yesterday's tutorial #1 (OpenCilk): programming language and interface, runtime supports [BL99, HLL10, SKL+15, SML19]
  - Yesterday's tutorial #2 (parallel trees): efficient tree algorithms both theoretically and practically [BFS16, SB19, SBLP19, SFB18]
- The parallel model in [CLRS, 3rd edition]

The multithreaded model in CLRS

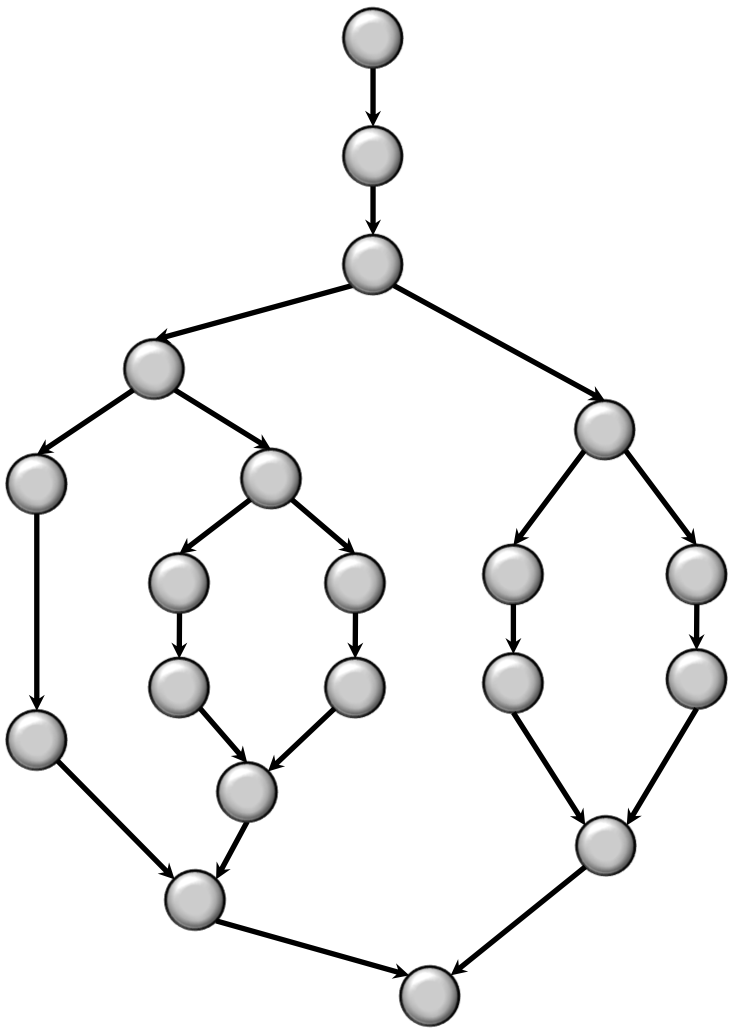
- Similar to a RAM, working on a shared-memory
- A `fork` instruction creates **two** subtasks that can be run in parallel
- After they finish, they `join` and continue
- Assumes **race-free**
  - Or allows some atomic operations such as `Test&Set`, `Compare&Swap`, and `Fetch&Add`



The multithreaded model in CLRS

- Similar to a RAM, working on a shared-memory
- A `fork` instruction creates **two** subtasks that can be run in parallel
- After they finish, they `join` and continue
- Assumes **race-free**
- An algorithm is measured by work (#operations) W and span/depth (longest dependence) D

Binary Fork-Join Model

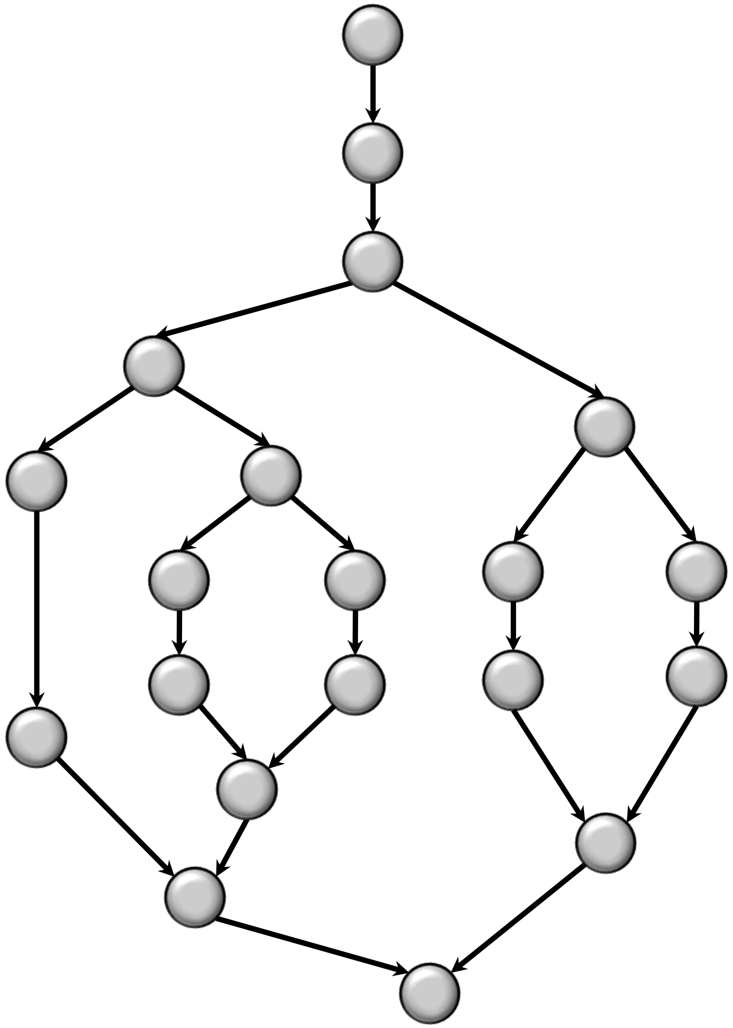


Why multithreaded models?

- Supported by existing libraries such as Cilk, OpenMP, TBB, Java Fork-Join, X10, TPL, Habanero
- Have good theoretical guarantee when mapping to hardware
  - Incur $O(PD)$ "steals" whp using a randomized work-stealing scheduler (under mild assumptions)

#processors   Span/depth

The computation is highly **asynchronous**



Why **revisit** multithreaded models?

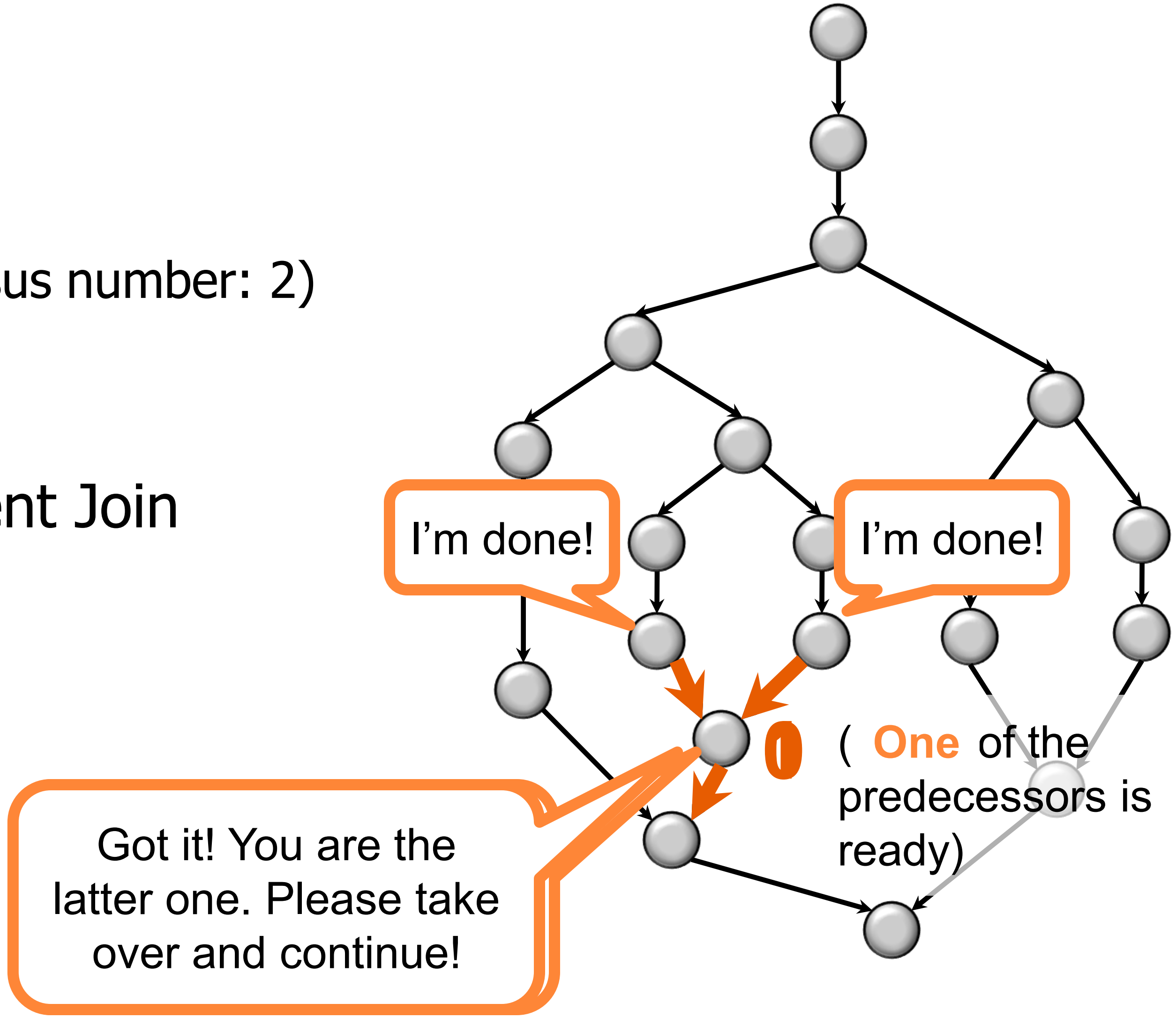
- Decision 1: binary forking vs. n-ary forking
- Decision 2: nested join vs. non-nested join
- Decision 3: race-free vs. assuming `Test&Set` vs. assuming `Compare&Swap` vs. assuming `Fetch&Add`

Rest of this talk

1. The binary-forking model
2. Algorithm design inspired in the asynchronous setting

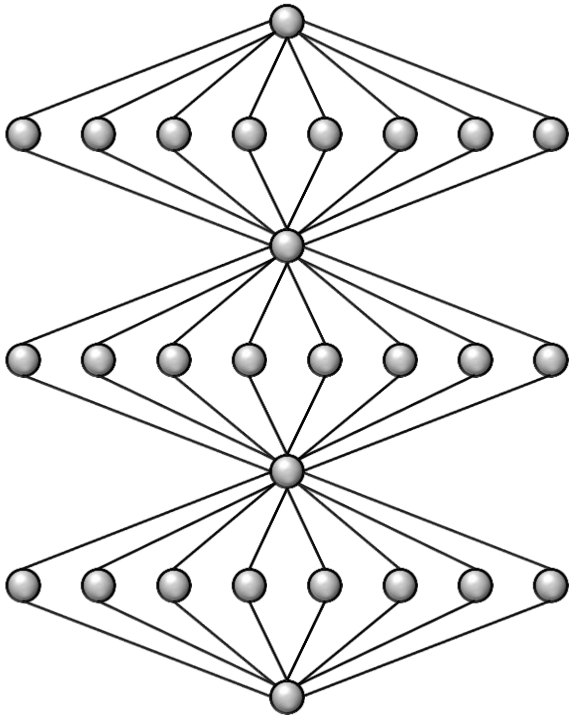
The binary-forking (BF) model

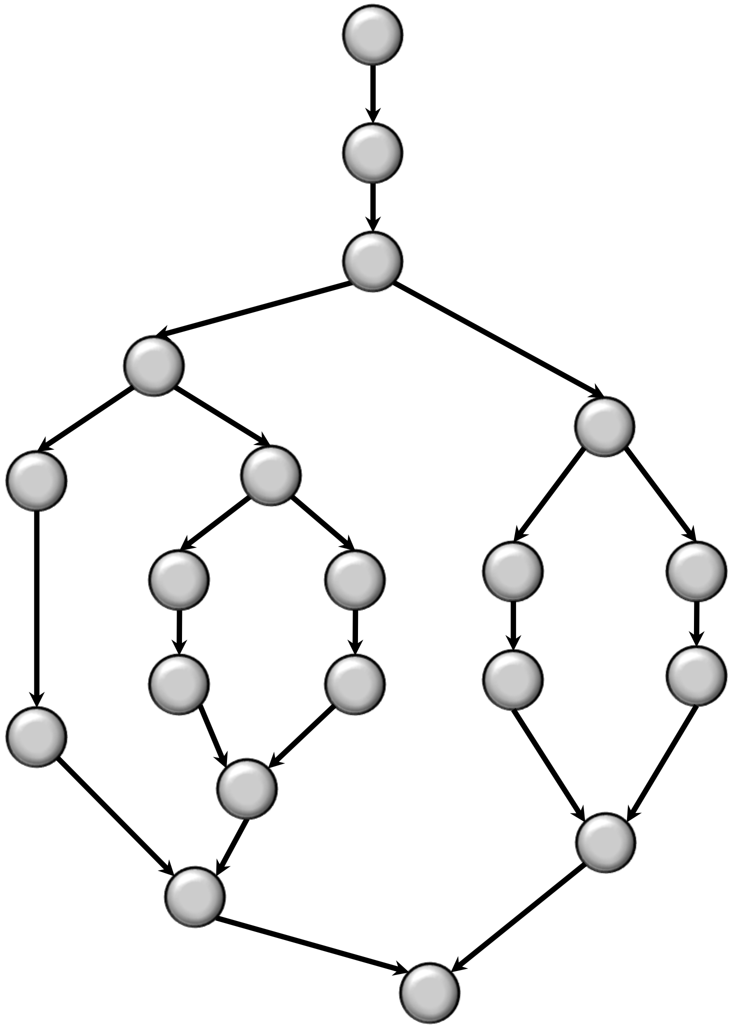
- `Fork` generates two subtasks
- Assumes `Test&Set`
  - Atomically set a bit flag from 0 to 1
  - Weakest consensus primitive (consensus number: 2)
- `Test&Set` can be used to implement Join



Q1: why **binary** when forking?

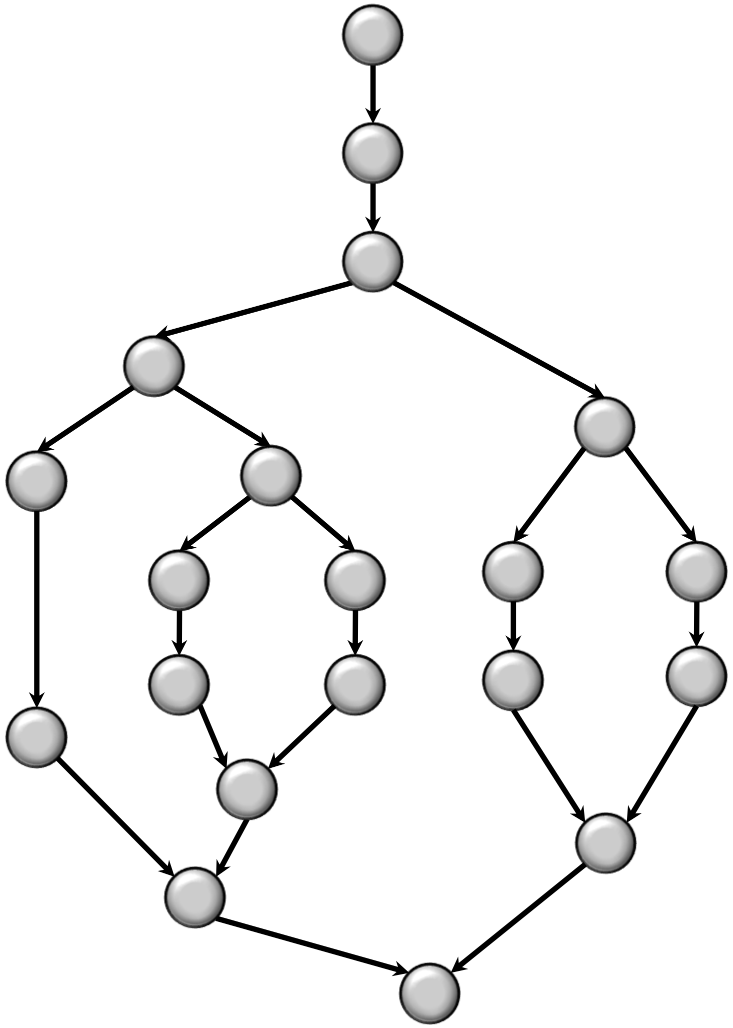
- In practice, that's how most existing software is implemented
- Arbitrary forking = global synchronization





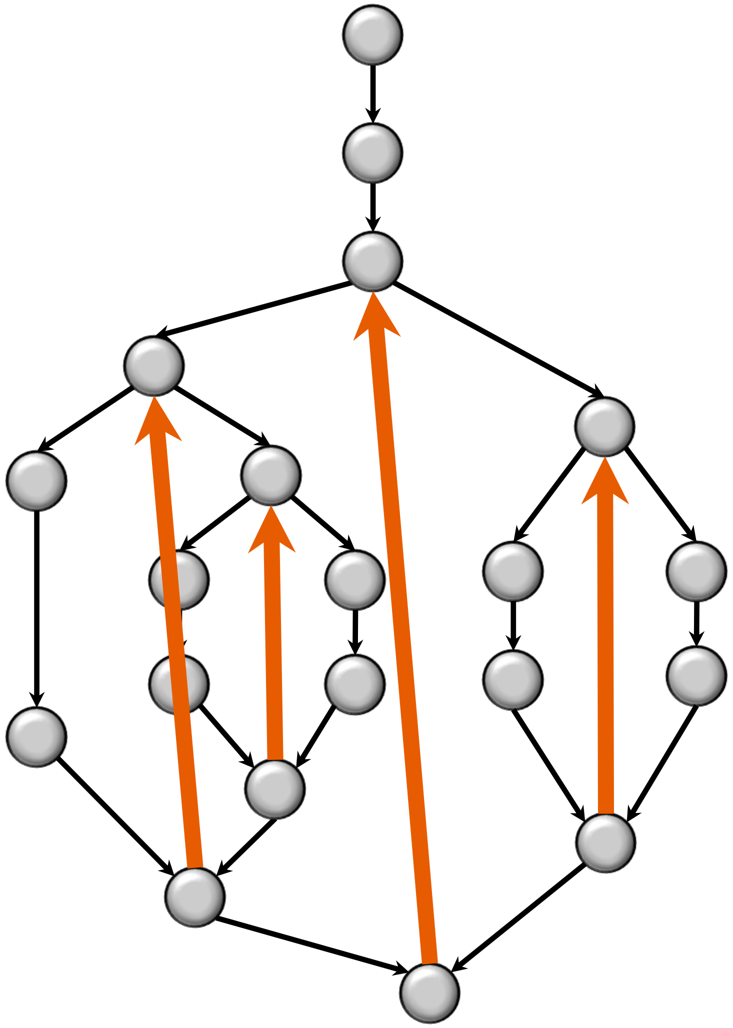
Q1: why **binary** when forking?

- In practice, that's how most existing software is implemented
- Arbitrary forking can abuse global synchronization (no difference as compared to the PRAM model)
- For dynamic work-stealing schedulers, most efficient scheduling results (e.g., **[BL99, ABP01, ABB02]**) hold only for **binary** forking
  - $O(PD)$ "steals" whp ($O\left(\frac{W}{P}+D\right)$ time)
  - Otherwise, the span bound can increase by $O(\log n)$



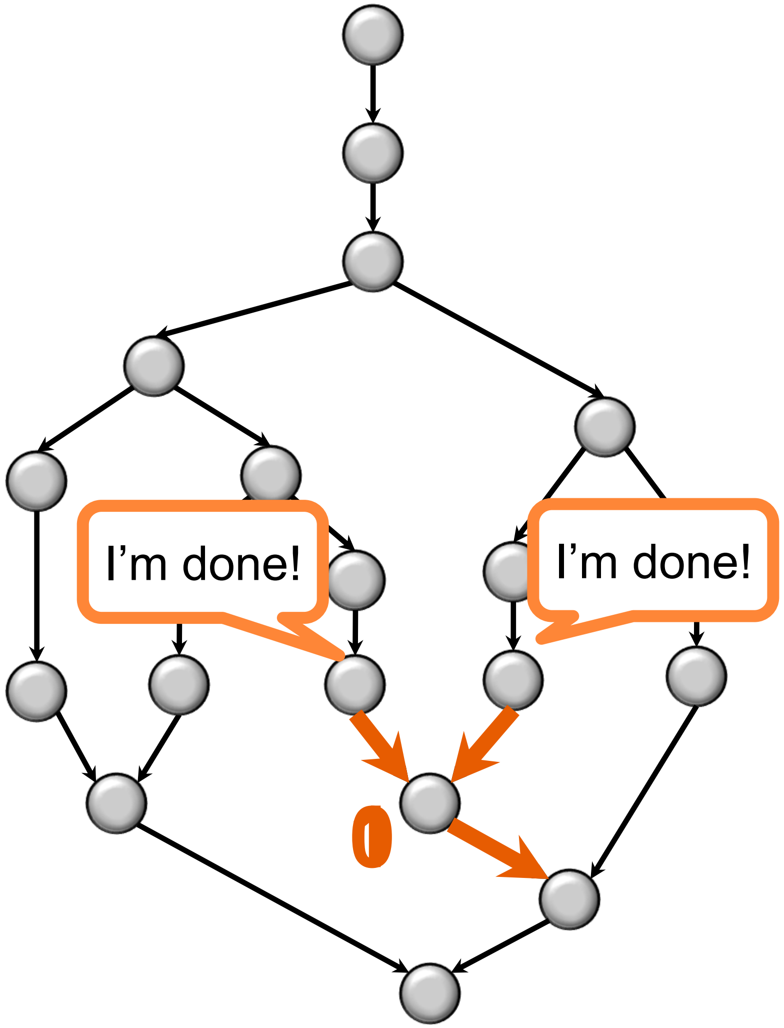
Q2: is **nested** join necessary?

- Easy to implement
  - Cilk: `cilk_spawn A(); B(); cilk_sync;`
  - Lambda: `par_do([&](){A();}, [&](){B();});`



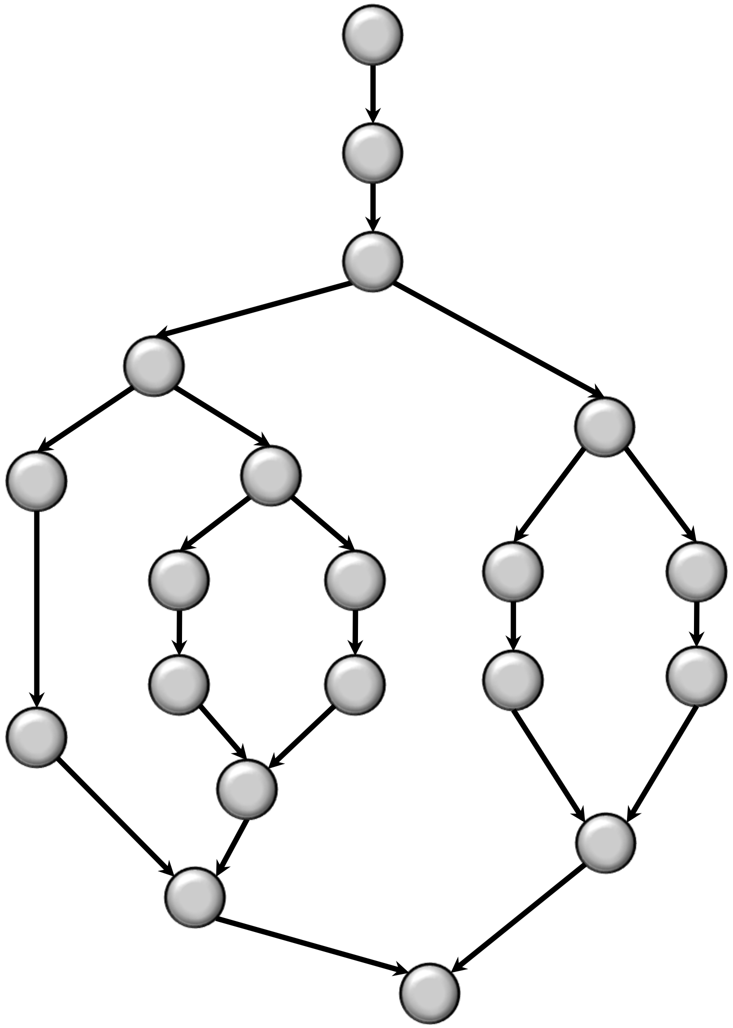
Q2: is **nested** join necessary?

- Easy to implement
  - Cilk: `cilk_spawn A(); B(); cilk_sync;`
  - Lambda: `par_do([&](){A();}, [&](){B();});`
- Scheduling results for work-stealing (e.g., **[ABP01, ABB02]**) do not require **nested** join
- In BF model, `Test&Set` allows non-nested join



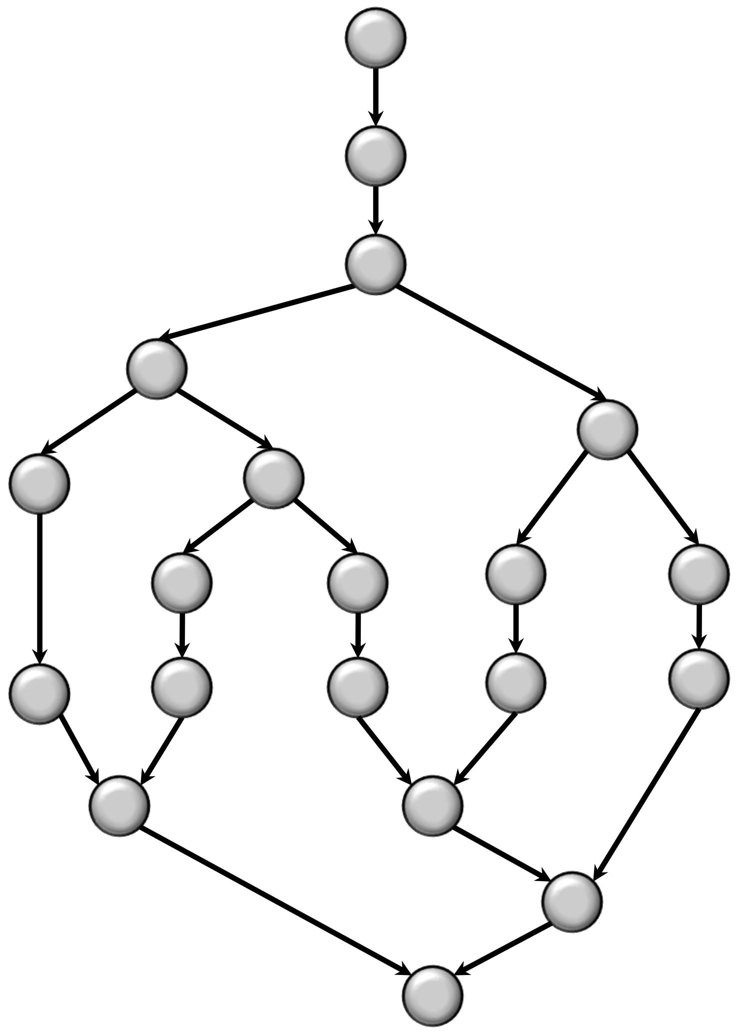
Q3: Why assume `Test&Set`?

- What is `Test&Set`?
  - Atomically set a bit flag from 0 to 1
  - Weakest consensus primitive (consensus number: 2)
- `Join` needs at least `Test&Set`
  - Two threads need to reach consensus
  - No stronger assumptions for real hardware
- Real architecture supports `Test&Set`



The binary-forking (BF) model

- `Fork` is binary
- Assumes `Test&Set`
  - Atomically set a bit flag from 0 to 1
  - Simplest consensus primitive (consensus number: 2)
- `Test&Set` can be used to implement Join (non-nested join)
- Theoretical scheduling guarantees still hold
  - Incur $O(PD)$ “steals” whp using a randomized work-stealing scheduler



The two derived models

- The binary fork-join model **[CLRS, 3rd edition]**
  - Assumes `Join` as a primitive, instead of `Test&Set`
  - Additional guarantees for race-free and determinism
- The binary-forking model with `CAS`
  - Assumes `Compare&Swap`, a stronger consensus primitive (consensus number: ∞)
  - Easier for practical programming

Outline of this talk

1. The binary-forking model

2. Algorithm design inspired in the asynchronous setting

Some PRAM algorithms are already optimal, but not all

- Lower bound: $\Omega(\log n)$ span for algorithm with $\Omega(n)$ work
- A parallel-for of size n has $O(\log n)$ span in the BF model
- Reduce, scan, filter are already optimal
- Cole's ingenious PRAM mergesort requires $O(\log n)$ rounds
  - Span on BF model is $O(\log n) \cdot O(\log n) = O(\log^2 n)$, the same as a normal mergesort
  - Asynchronous samplesort **[BGS10, BAD20]** with $o(\log^2 n)$ span is faster in practice

Optimal parallel algorithms in the BF model

- $O(\log n)$ span ➔ none or a constant number of global synchronization
- New results for all these fundamental problems
- Surprisingly, these algorithms are all quite simple

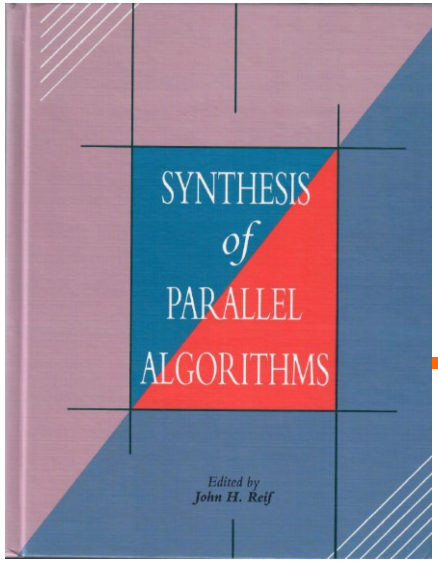
Problem	Work	Span/ depth
List contraction	$O(n)$	$O(\log n)^*$
Comparison sort	$O(n \log n)^\uparrow$	$O(\log n)^*$
Semisort	$O(n)^\uparrow$	$O(\log n)^*$
Random permutation	$O(n)^\uparrow$	$O(\log n)^*$
Range minimum query	$O(n)$	$O(\log n)$
Tree contraction	$O(n)$	$O(\log n)^*$
Ordered-set Operations (Union, intersect, diff)	$O\left(m \log\left(\frac{n}{m}+1\right)\right)$	$O(\log n)$

$^\uparrow$ expected, * with high probability

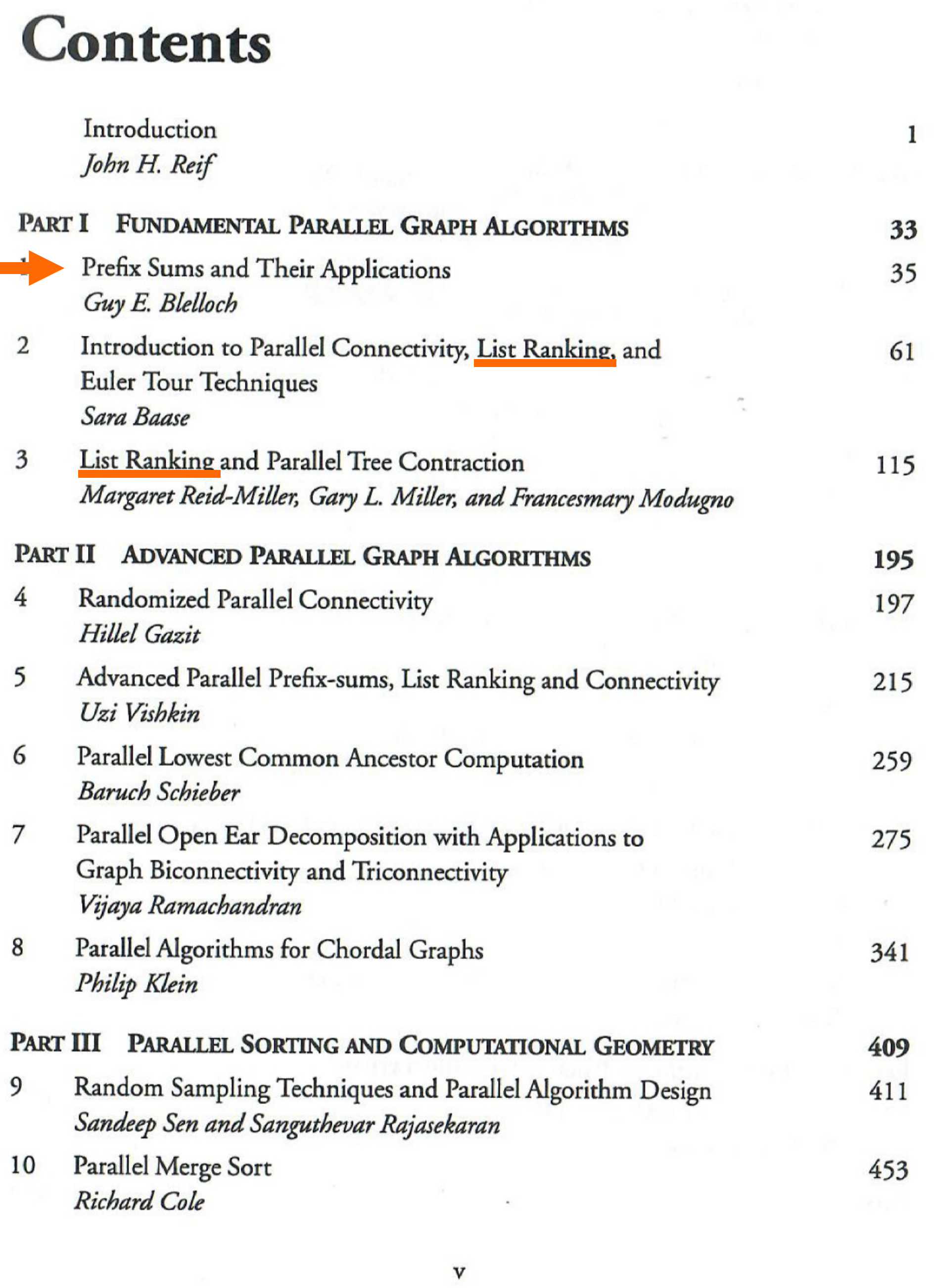
List ranking

- Prefix sum (scan)

[1, 2, 3, 4, 5] → [1, 3, 6,

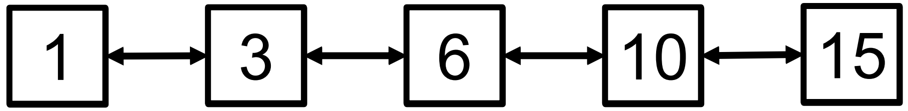


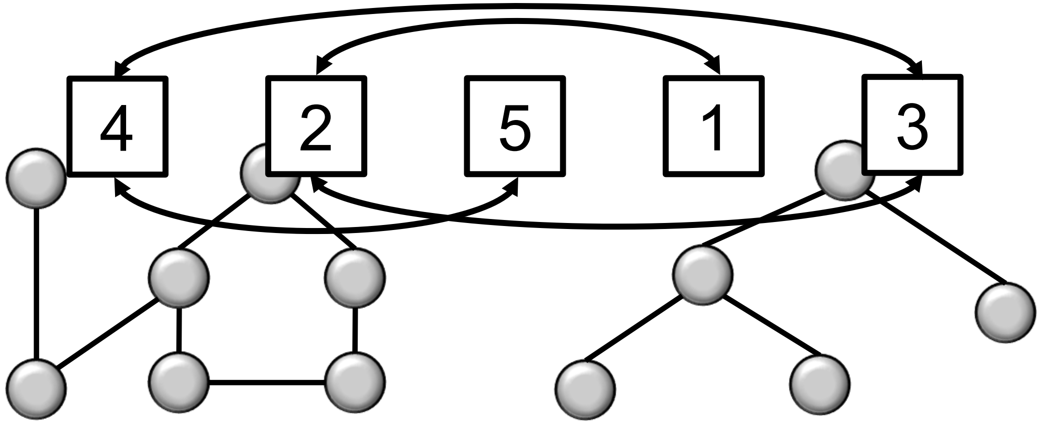
Synthesis of Parallel Algorithms



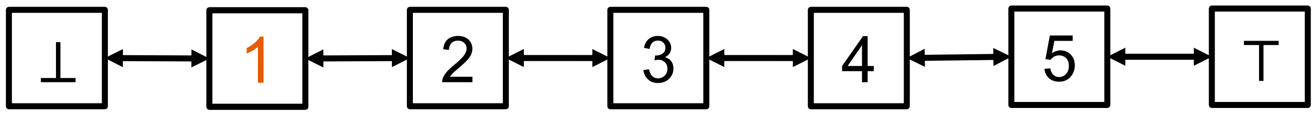
Contents

- List ranking

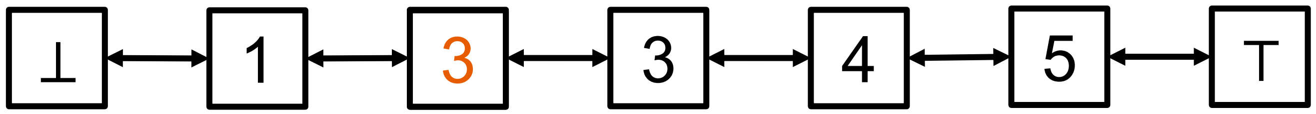




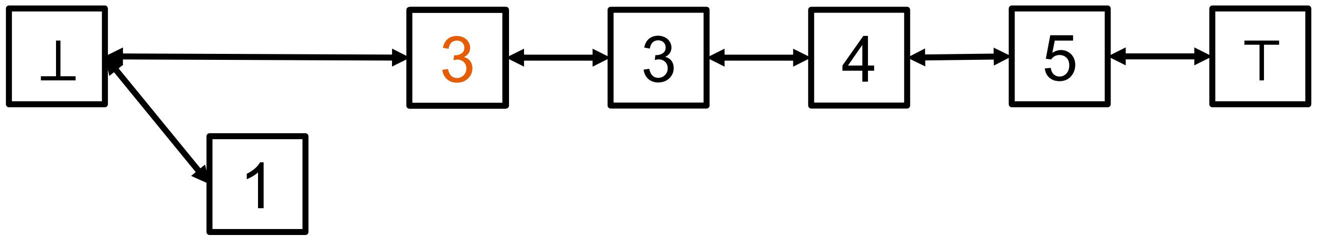
The splice operation for list ranking



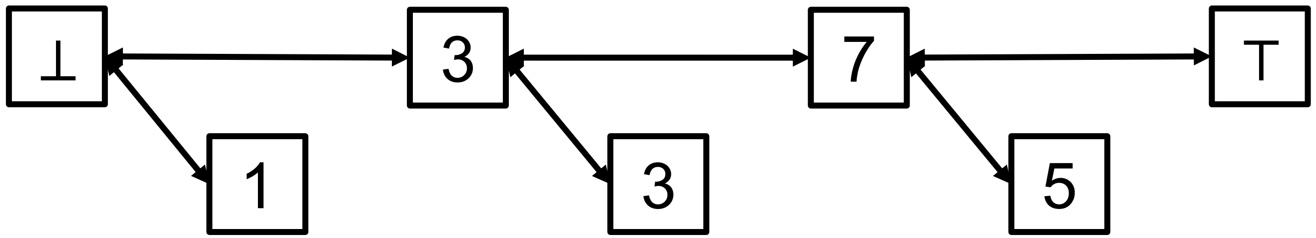
The splice operation for list ranking



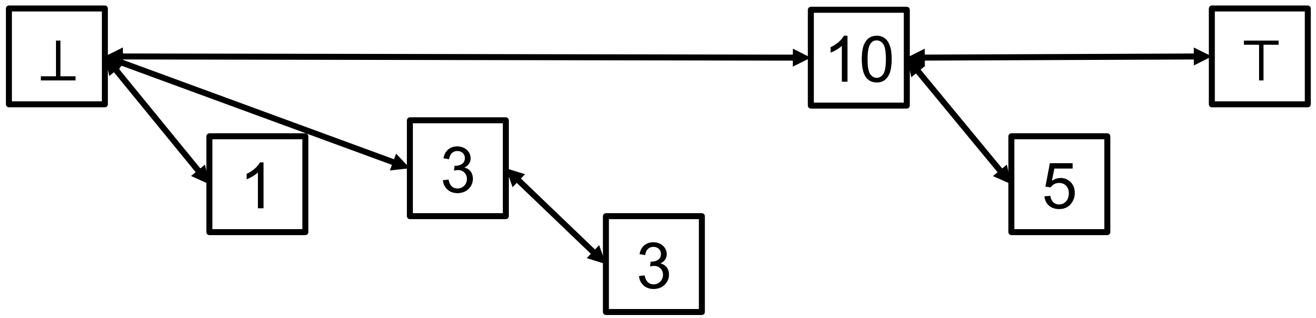
The splice operation for list ranking



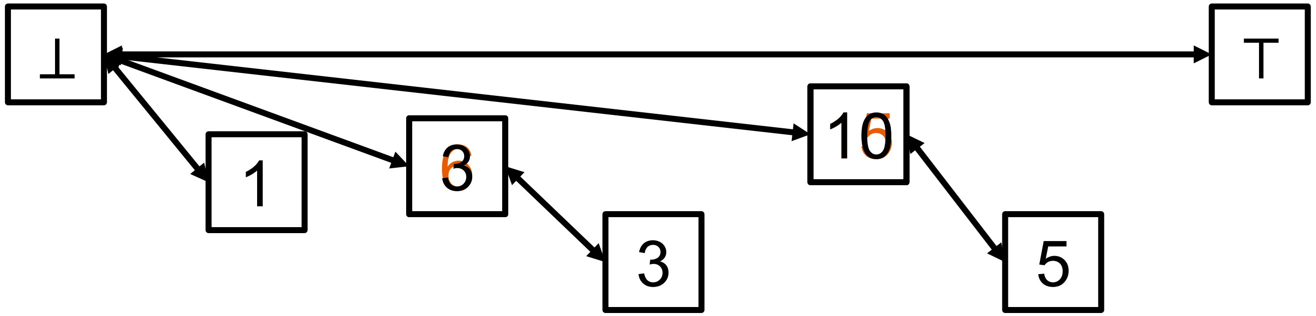
The splice operation for list ranking



The splice operation for list ranking



The splice operation for list ranking



List contraction + Reconstruction = List Ranking

Use splice operation to contract the list and generate a tree with shallow (ideally $O(\log n)$) depth

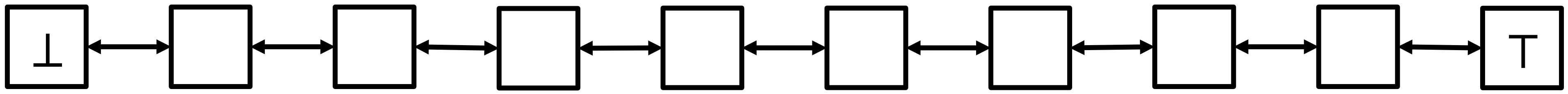
List contraction

- Have been studied for decades **[AM13, Baase93, CV86, JáJá92, JLS14, KR90, Ranade98, RMM93, SGB⁺15, Vishkin84, Vishkin93, Wyllie79]**
  - The optimal ones (on PRAM) are complicated
  - The practical ones are not theoretically efficient
  - All existing algorithms requires $\Omega(\log n)$ rounds of global synchronization

- Goal: simple, both theoretically and practically efficient algorithm, using no global synchronization

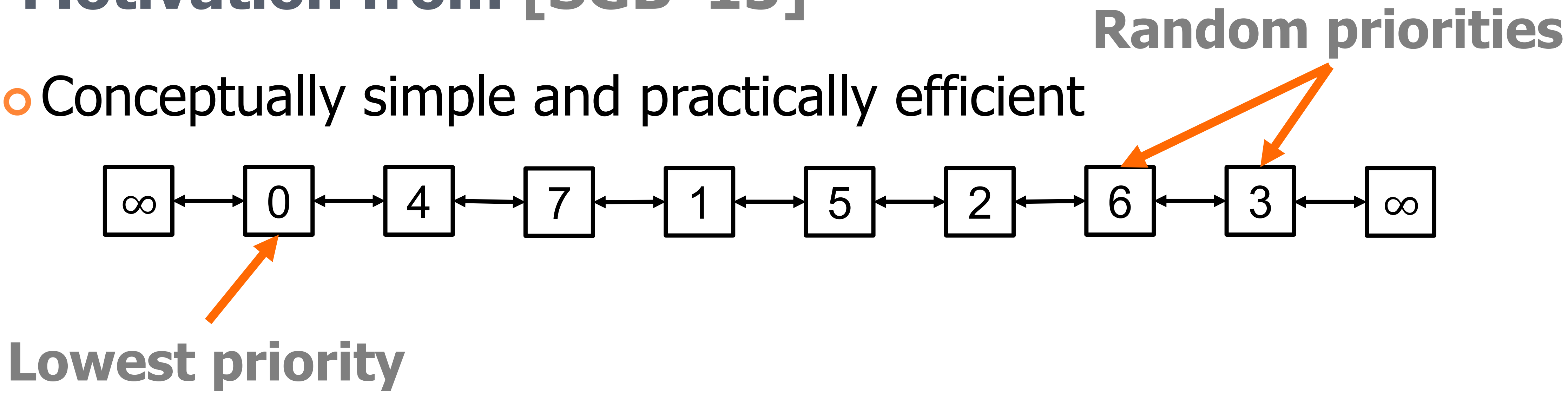
Motivation from [SGB+15]

- Conceptually simple and practically efficient



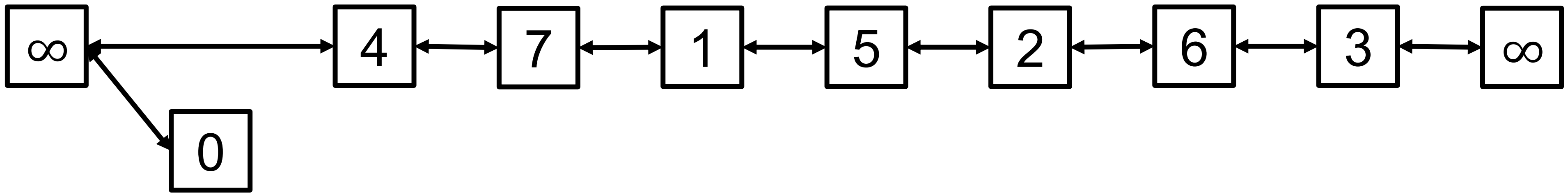
Motivation from [SGB+15]

- Conceptually simple and practically efficient



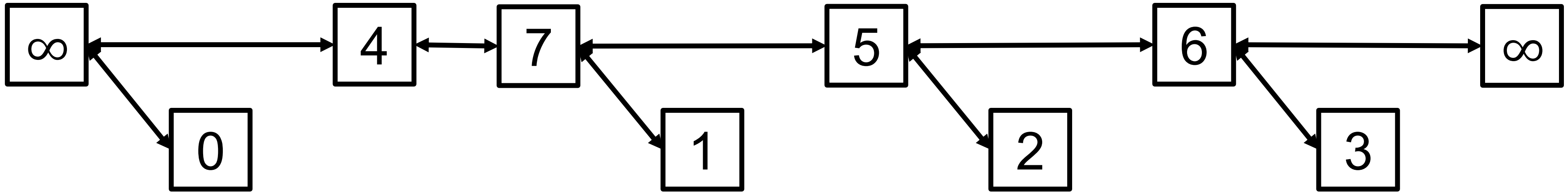
Motivation from [SGB+15]

- Conceptually simple and practically efficient



Motivation from [SGB+15]

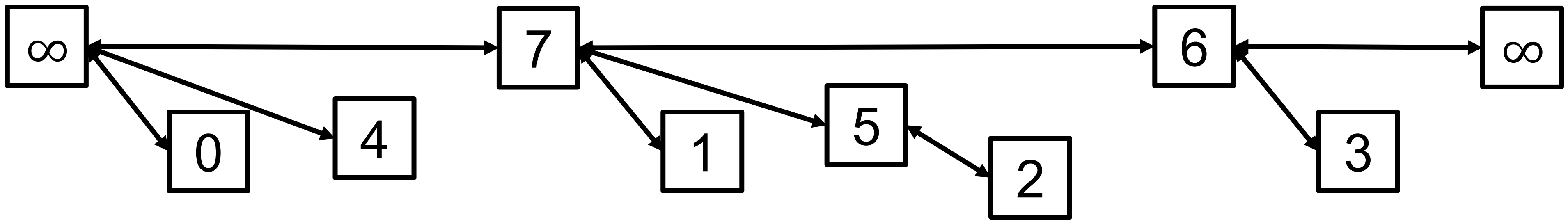
- Conceptually simple and practically efficient



Round 1

Motivation from [SGB+15]

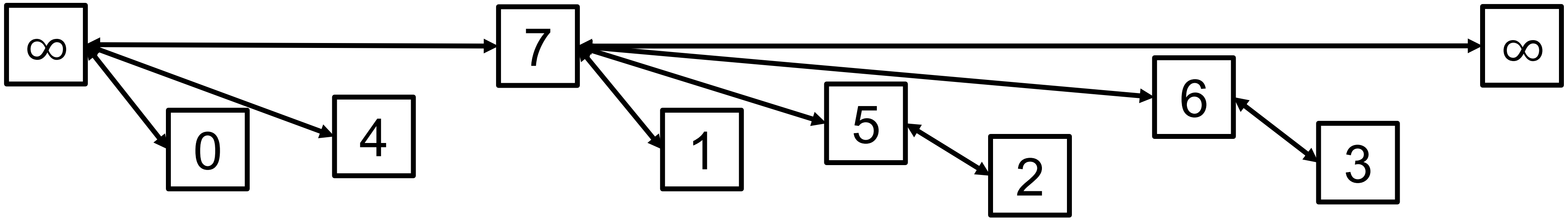
- Conceptually simple and practically efficient



Round 2

Motivation from [SGB+15]

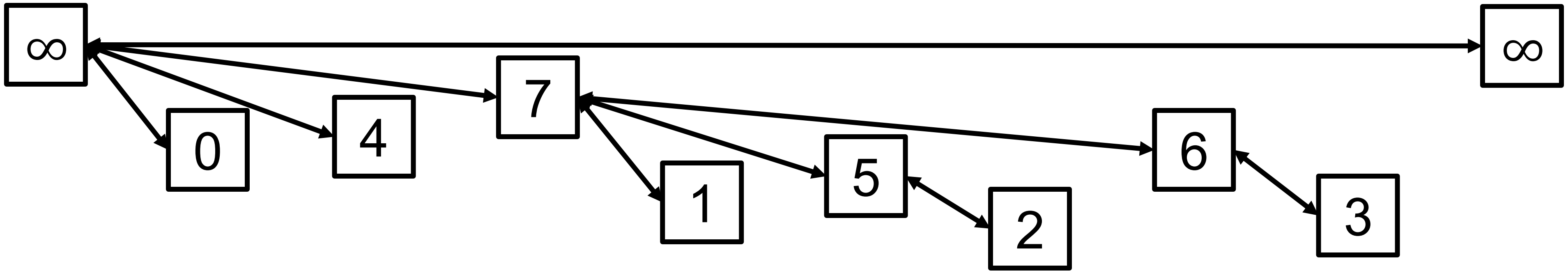
- Conceptually simple and practically efficient



Round 3

Motivation from [SGB+15]

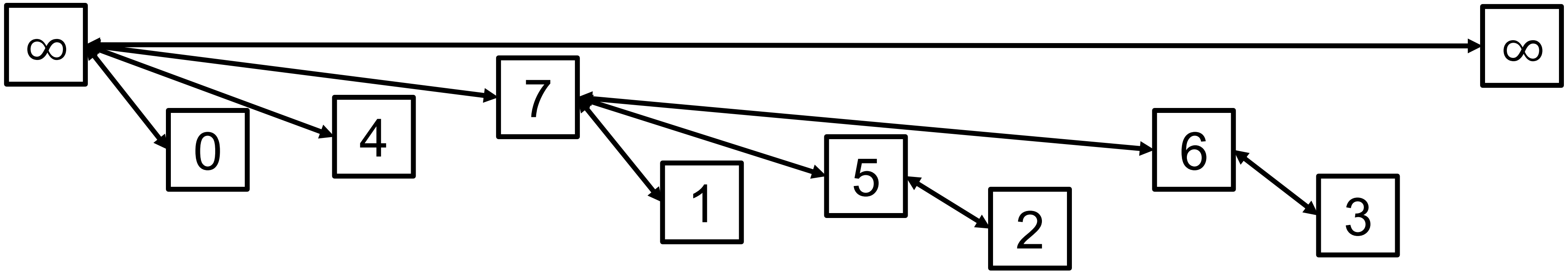
- Conceptually simple and practically efficient



Round 4

Motivation from [SGB+15]

- Conceptually simple and practically efficient



Give a random priority to each node

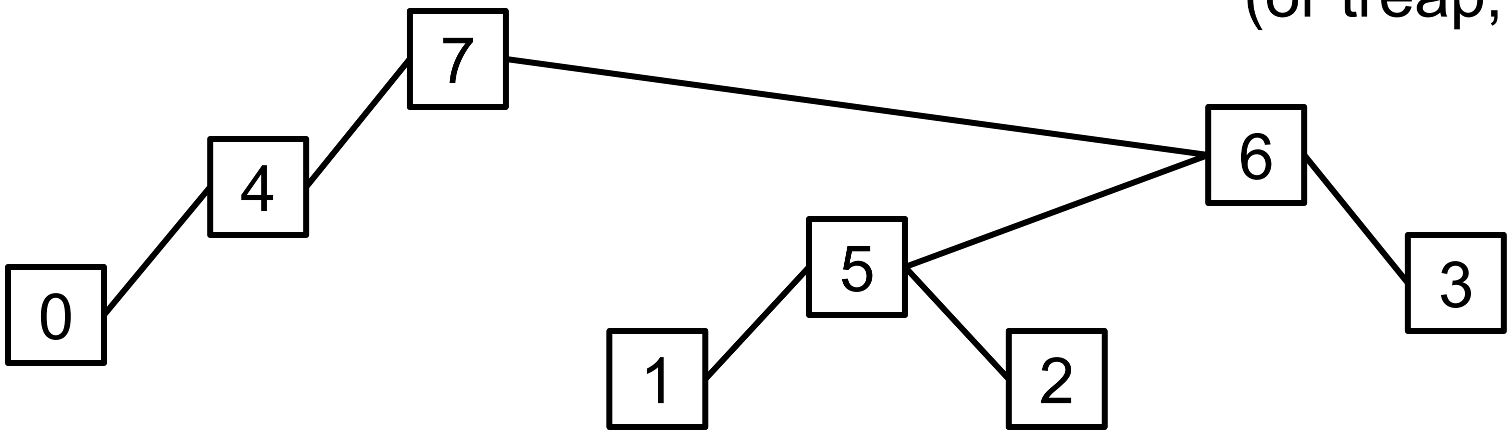
While not fully contracted

Contract all feasible nodes in parallel

Pack the rest of the nodes

Dependency between splices

Tree height is $\Theta(\log n)$ *whp* [SGB+15], the same as a randomized BST (or treap, quicksort)



Give a random priority to each node

While not fully contracted

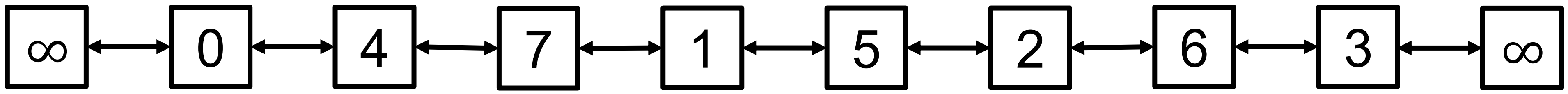
Contract all feasible nodes in parallel

Pack the rest of the nodes

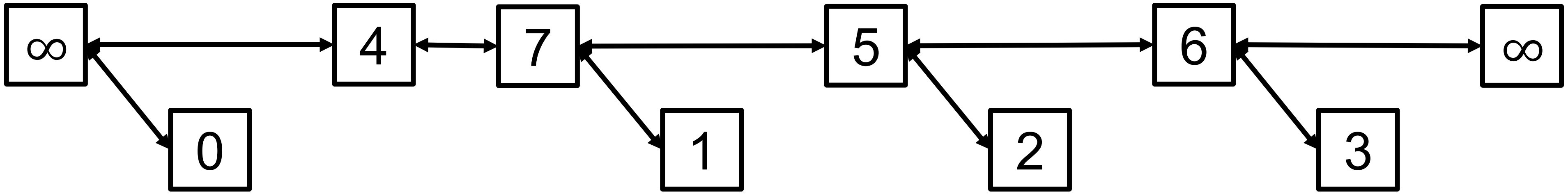
Disadvantages:

- $\Omega(\log n)$ rounds of synchronization
- Need to pack in every round

Asynchrony in splicing

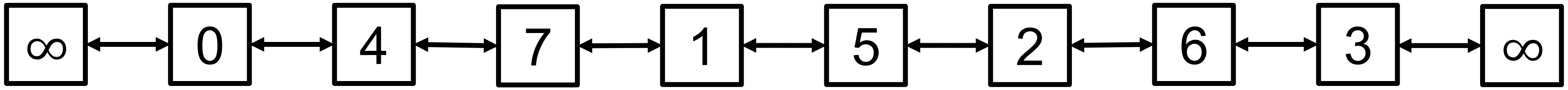


Asynchrony in splicing

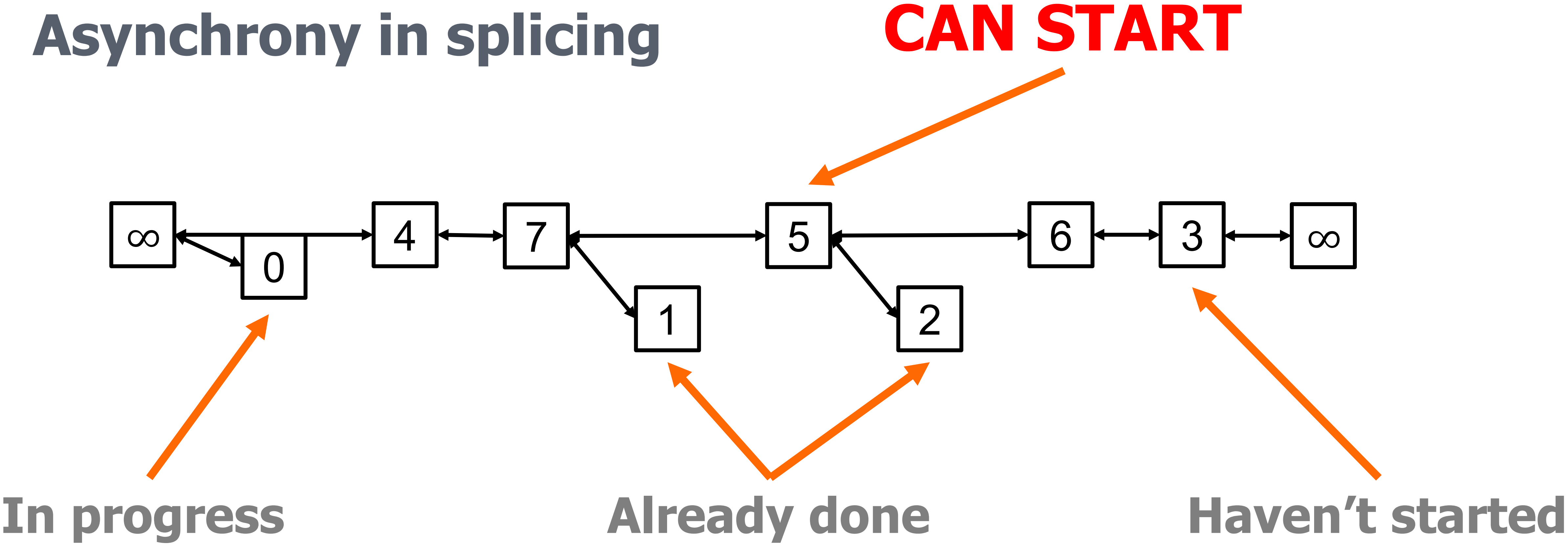


Round 1

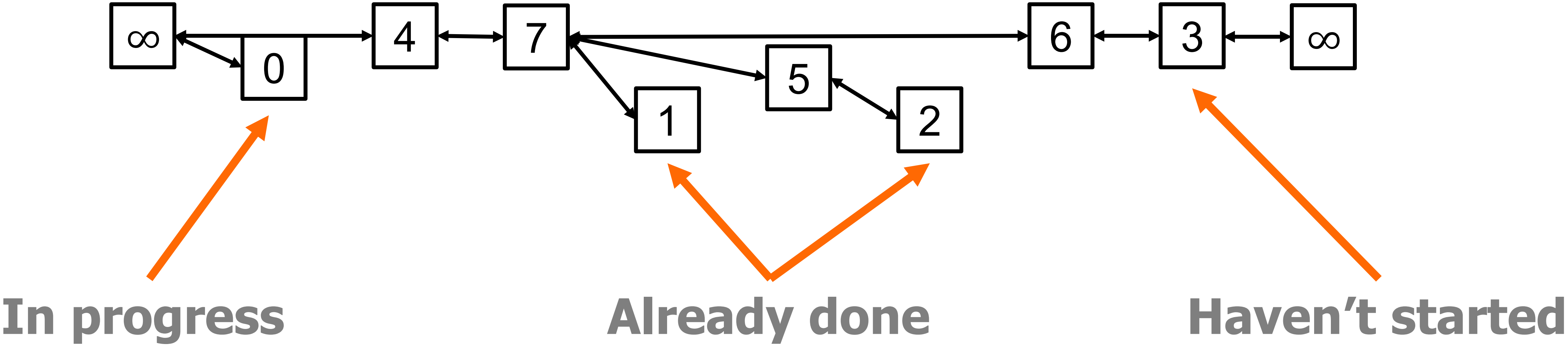
Asynchrony in splicing



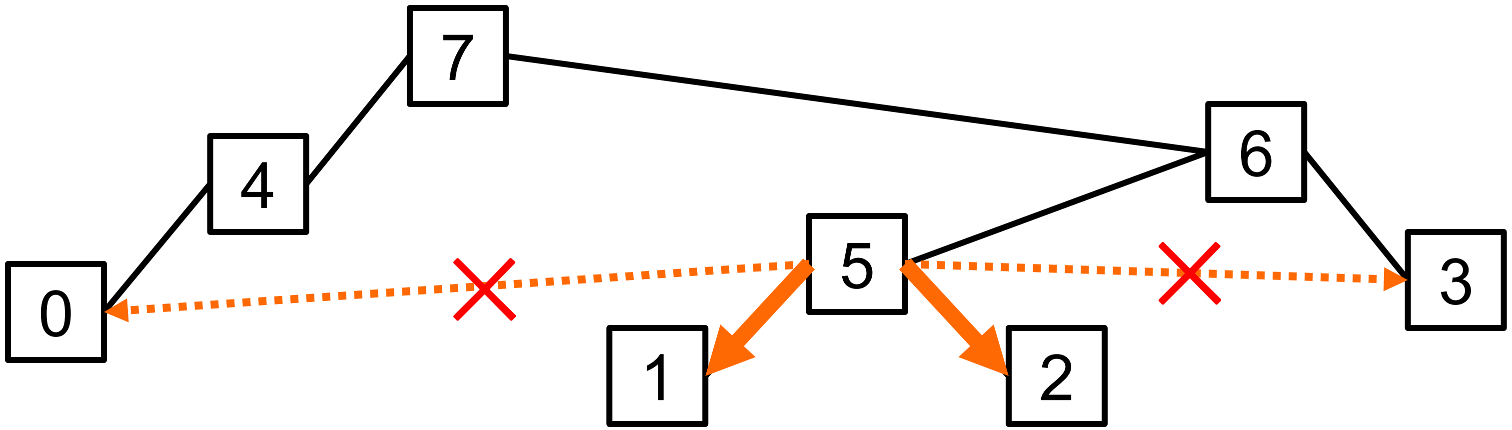
Asynchrony in splicing



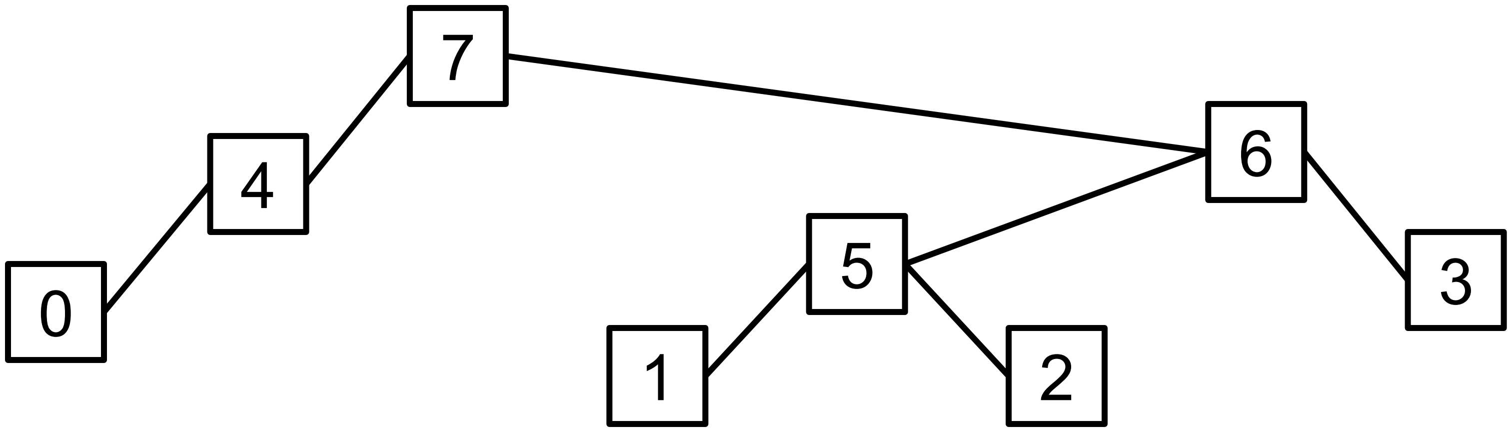
Asynchrony in splicing



New list contraction algorithm



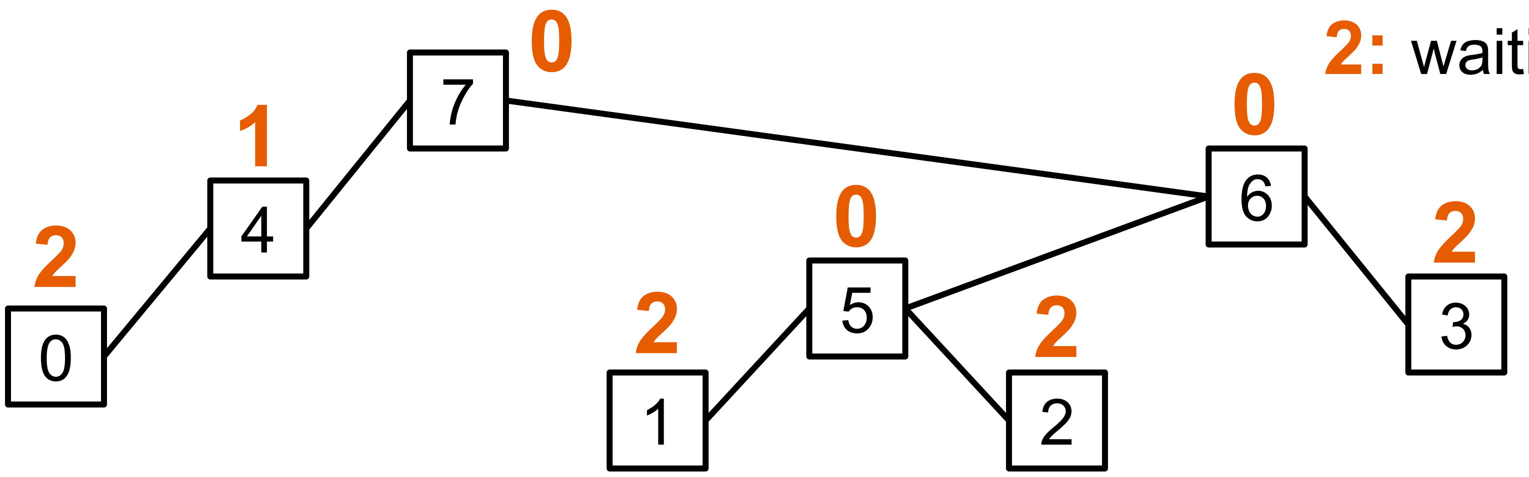
New list contraction algorithm



set the flag for each node
parallel-for each node v **do**
    if v is leaf **then**
        while list is not empty **do**
            splice v out
            $v \leftarrow v$.prev or v.next with smaller priority
            if `Test&Set`(v.flag) then break

New list contraction algorithm

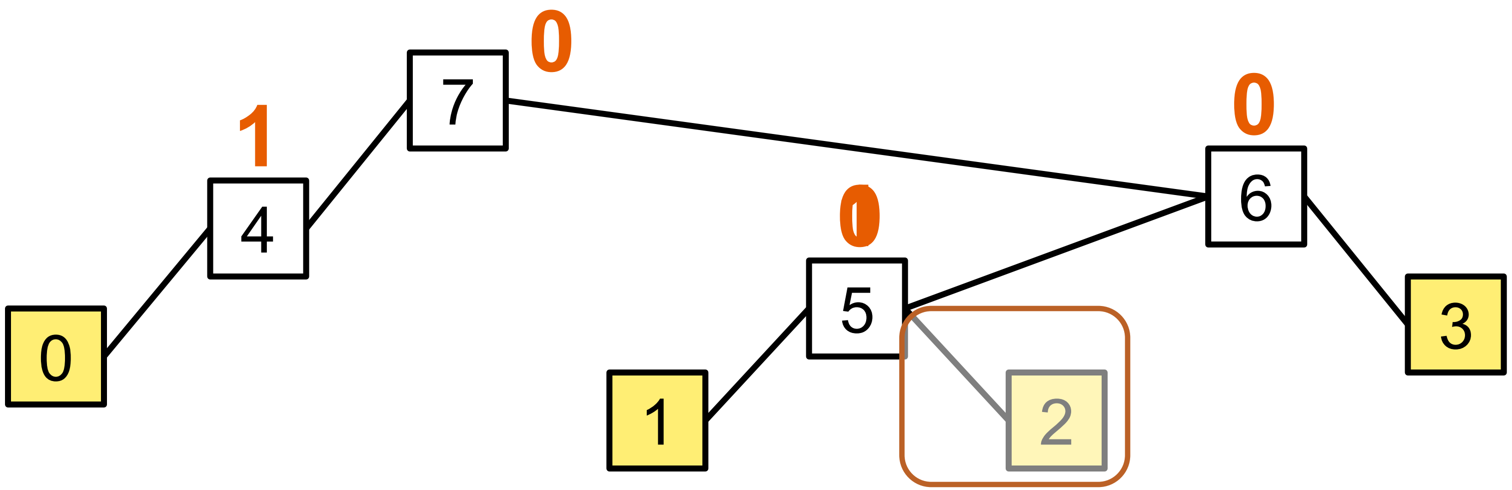
0: waiting for two children
1: waiting for one child
2: waiting for no children



set the flag for each node
parallel-for each node v **do**
  if v is leaf **then**
    while list is not empty **do**
      splice v out
      $v \leftarrow v$.prev or v.next with smaller priority
      if `Test&Set`(v.flag) then break

New list contraction algorithm

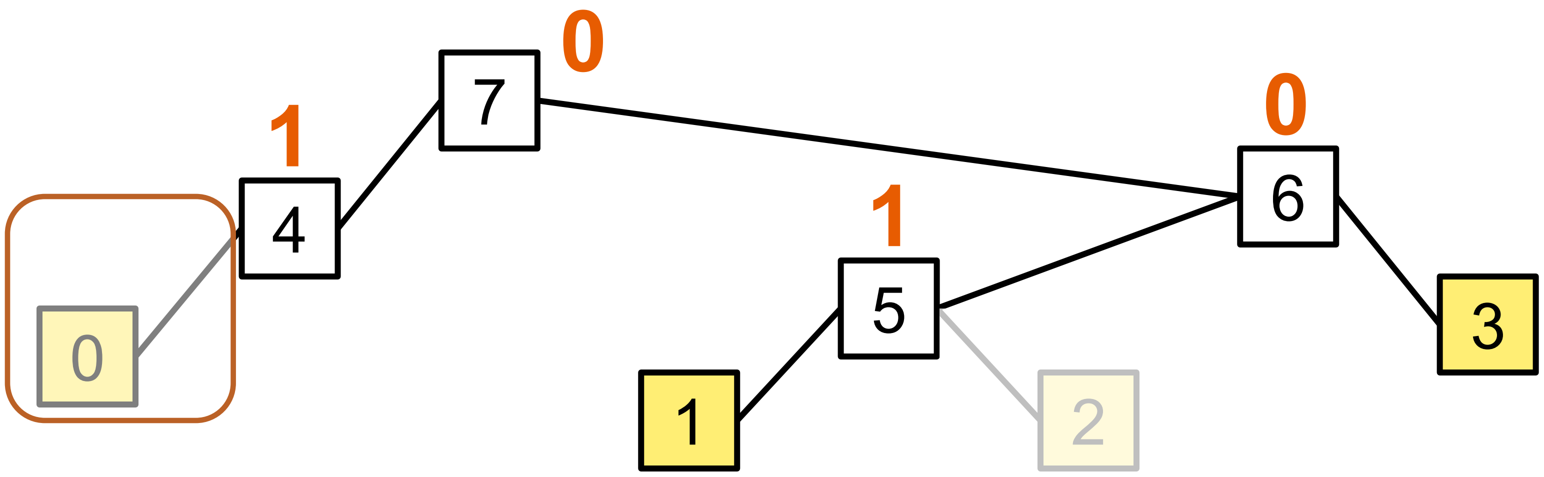
0: waiting for two children
1: waiting for one child



set the flag for each node
parallel-for each node v **do**
    if v is leaf **then**
        while list is not empty **do**
            splice v out
            $v \leftarrow v$.prev or v.next with smaller priority
            if `Test&Set`(v.flag) then break

New list contraction algorithm

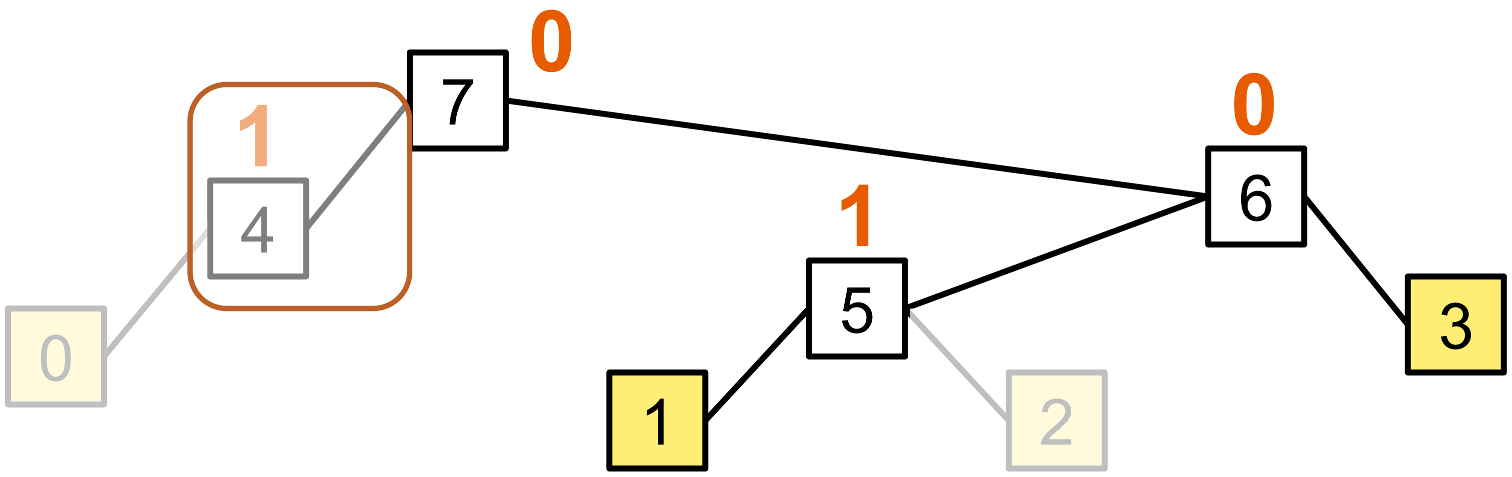
0: waiting for two children
1: waiting for one child



set the flag for each node
parallel-for each node v **do**
    if v is leaf **then**
        while list is not empty **do**
            splice v out
            $v \leftarrow v$.prev or v.next with smaller priority
            if `Test&Set`(v.flag) then break

New list contraction algorithm

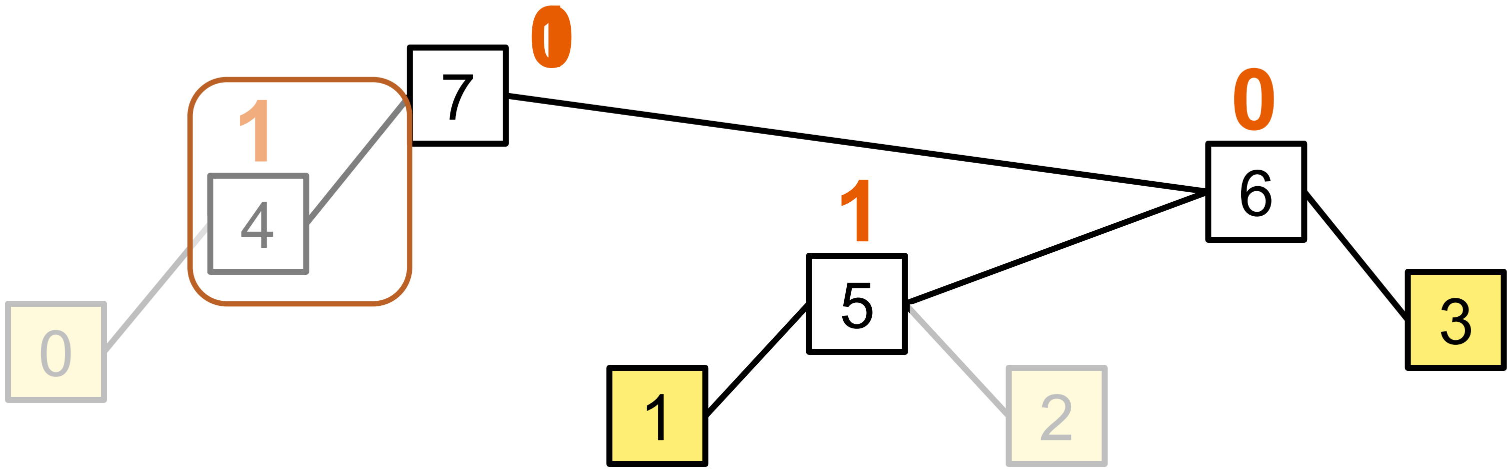
0: waiting for two children
1: waiting for one child



set the flag for each node
parallel-for each node v **do**
  if v is leaf **then**
    while list is not empty **do**
      splice v out
      $v \leftarrow v$.prev or v.next with smaller priority
      if Test&Set(v.flag) then break

New list contraction algorithm

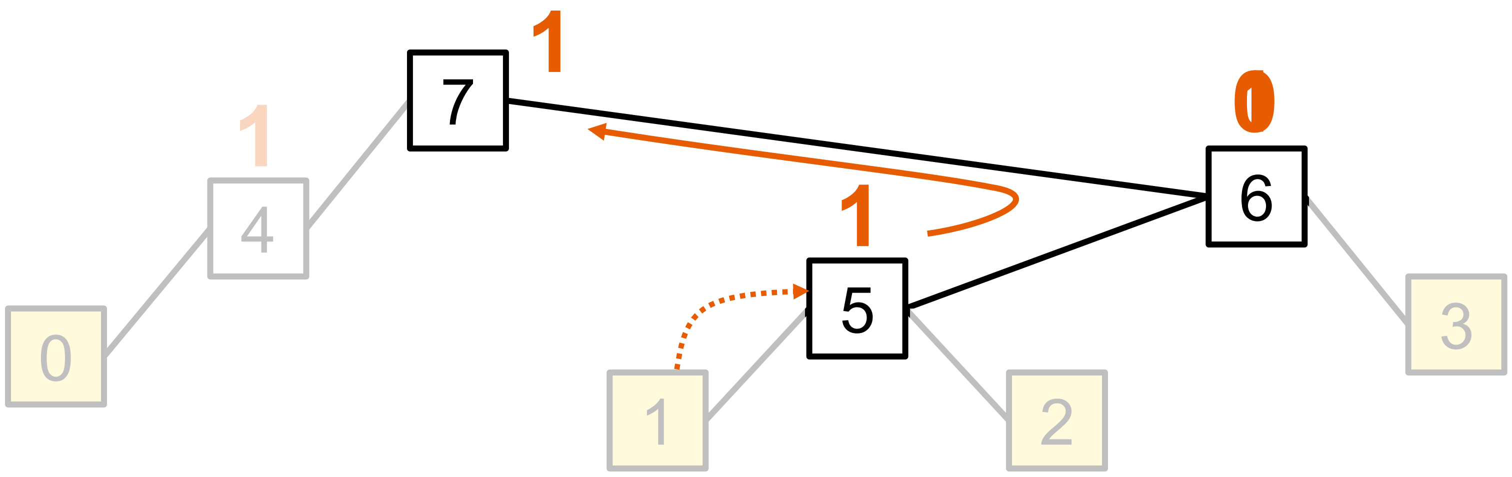
0: waiting for two children
1: waiting for one child



set the flag for each node
parallel-for each node v **do**
    if v is leaf **then**
        while list is not empty **do**
            splice v out
            $v \leftarrow v$.prev or v.next with smaller priority
            if `Test&Set`(v.flag) then break

New list contraction algorithm

0: waiting for two children
1: waiting for one child



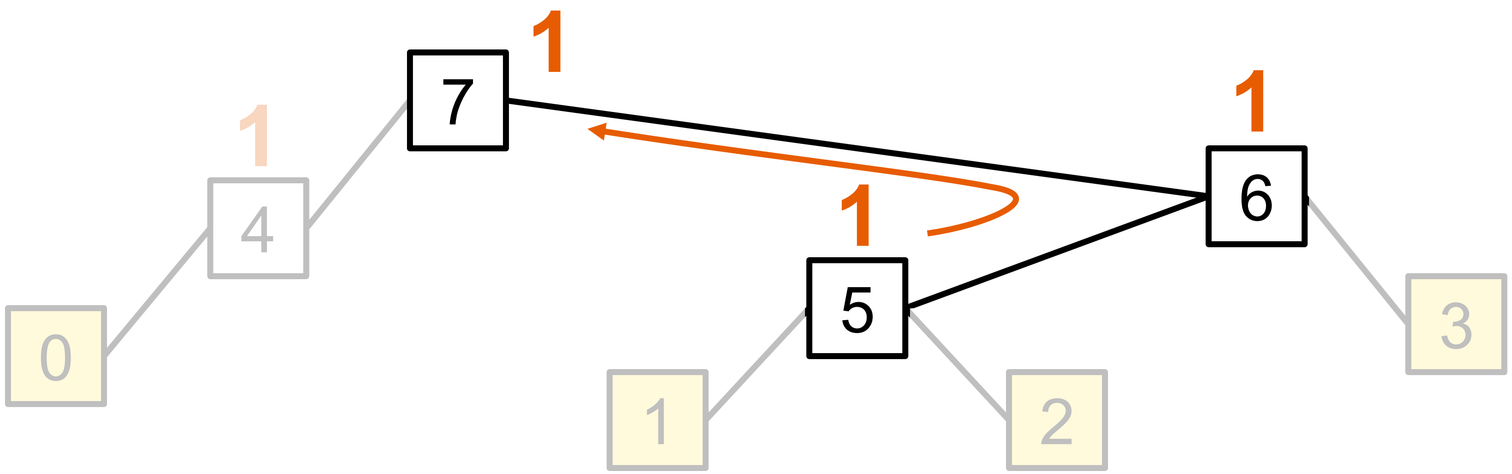
set the flag for each node
parallel-for each node v **do**
  if v is leaf **then**
    while list is not empty **do**
      splice v out
      $v \leftarrow v$.prev or v.next with smaller priority
      if `Test&Set`(v.flag) then break

No packing

No global synchronization

New list contraction algorithm

0: waiting for two children
1: waiting for one child



set the flag for each node
parallel-for each node v **do**
    if v is leaf **then**
        while list is not empty **do**
            splice v out
            $v \leftarrow v$.prev or v.next with smaller priority
            if `Test&Set`(v.flag) then break

Worst-case

Work: $O(n)$

Span: $O(\log n)$ *whp*

Extremely simple to implement (<20 lines)

set the flag for each node
parallel-for each node v **do**
  if v is leaf **then**
    while list is not empty **do**
      splice v out
      $v \leftarrow v$.prev or v.next with
        smaller priority
      if `Test&Set`(v.flag) then break

```
void listRanking(node *A, intT n) {
  char* R = new char[n];
  parallel_for (intT i=0; i < n; i++)
    R[i] = (A[i].pri < A[A[i].prev].pri) || (A[i].pri < A[A[i].next].pri);
  parallel_for (intT i=0; i < n; i++) if (R[i] <= 1) {
    intT c = i, prev = A[c].prev, next = A[i].next;
    if ((A[i].pri < A[prev].pri) && (A[i].pri < A[next].pri)) {
        while (A[c].prev != n || A[c].next != n) {
            A[next].prev = prev;
            A[prev].next = next;
            R[c] = 2;
            c = A[prev].pri < A[next].pri ? prev : next;
            if (utils::CAS(&R[c], (char)0, (char)1)) break;
            prev = A[c].prev; next = A[c].next;
        }
    }
  }
  delete[] R;
}
```

More efficient than the highly optimized code in PBBS from [SGB+15]

Input size (million)	Running time (s)		Improved by
	New algorithm	PBBS version	
10	0.039	0.082	110%
20	0.094	0.16	70%
50	0.23	0.38	65%
100	0.51	0.72	41%
200	1.11	1.53	38%
500	2.85	3.85	35%

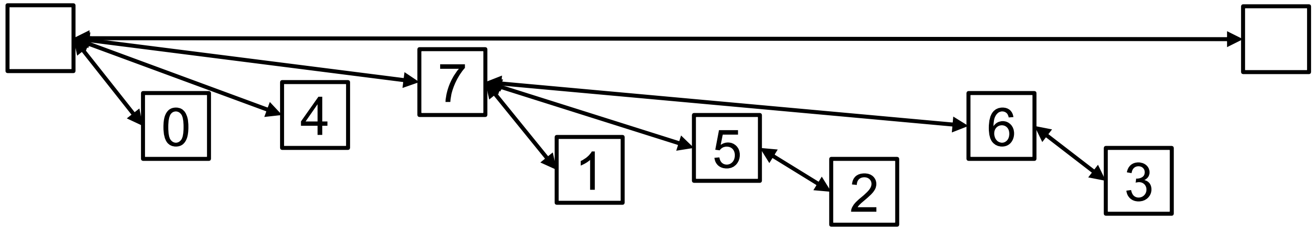
g++ with Cilk
on 72 cores

Saved by avoiding **synchronization**
(more when input is small)

Saved by avoiding **packing**
(consistent for all cases)

Summary for the new list contraction algorithm

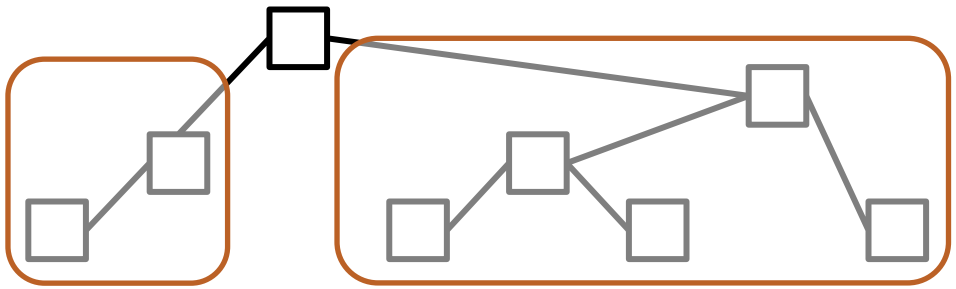
set the flag for each node
parallel-for each node v **do**
  if v is leaf **then**
    while list is not empty **do**
      splice v out
      $v \leftarrow v$.prev or v.next with smaller priority
      if `Test&Set`(v.flag) then break



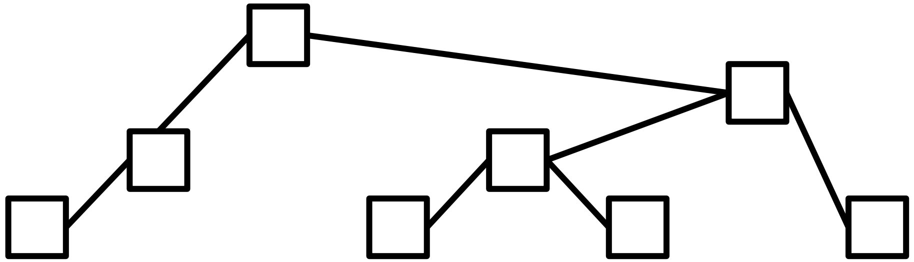
- Simple both conceptually and practically (rely on `Test&Set`)
- Efficient both theoretically and practically
- Highly asynchronous and not round based

Why so simple? Anything more general?

- Challenge in parallelism: dependencies
  - Classic solution: divide-and-conquer, round-based

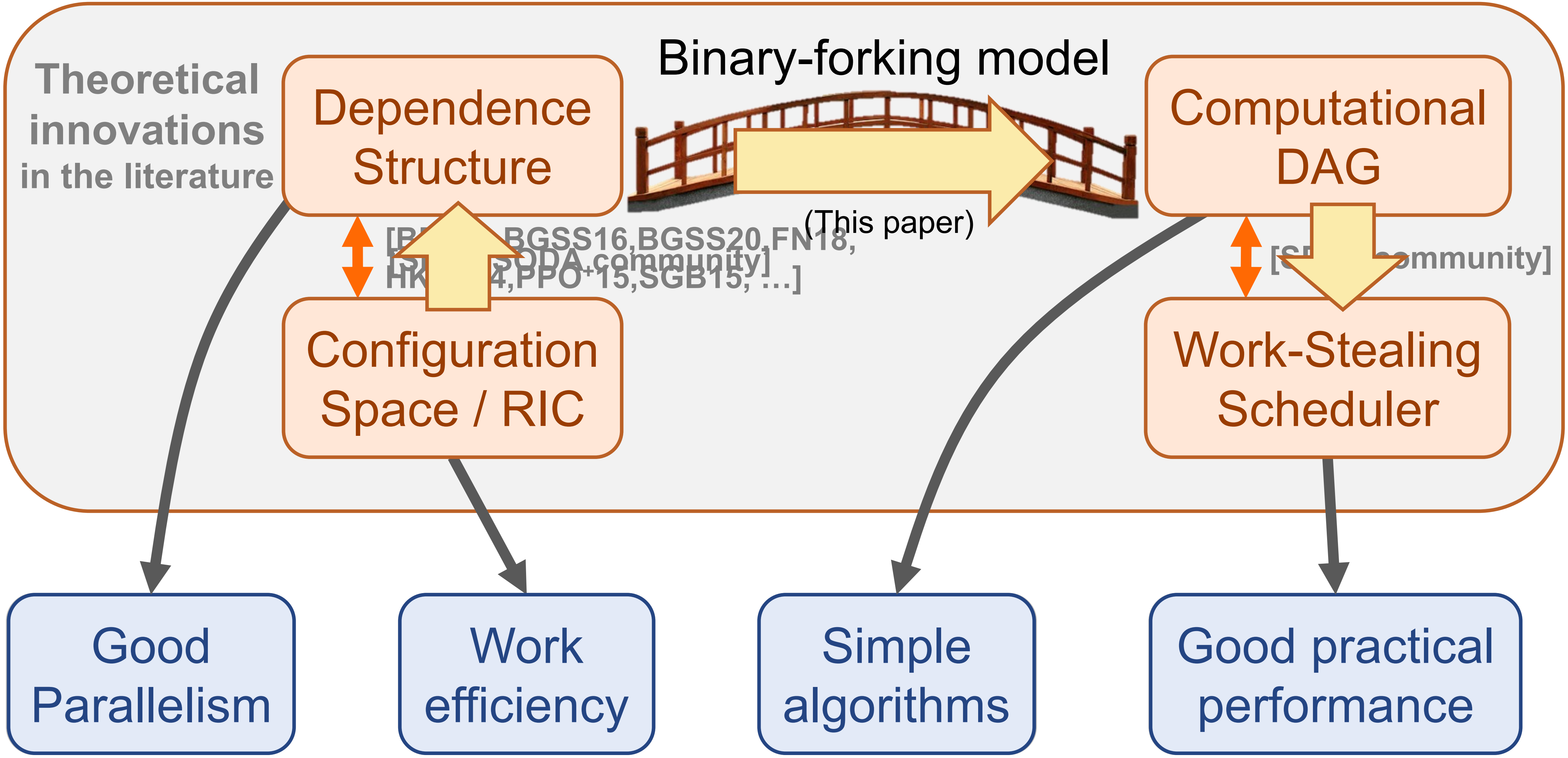


Why so simple? Anything more general?



- Challenge in parallelism: dependencies
  - Classic solution: divide-and-conquer, round-based
- What if we directly convert dependence structure to the computational DAG?
  - The rest of the work is done by the work-stealing scheduler!
  - Works for constant dependencies per node via `Test&Set`
  - Simple algorithms: List contraction, tree contraction, random permutation, sequence algorithms (e.g., RMQ), dynamic programs
- Idea combined with configuration space [Mulmuley 1994] solved **open problems**: incremental Delaunay triangulation [JACM 2020], convex hull [SPAA 2020] (a general framework for analyzing this type of algorithms)

A new angle for (asynchronized) parallel algorithms



Other optimal parallel algorithms in the BF model

Problem	Work	Span/ depth
List contraction	$O(n)$	$O(\log n)^*$
Sorting	$O(n \log n)^{\uparrow}$	$O(\log n)^*$
Semisort	$O(n)^{\uparrow}$	$O(\log n)^*$
Random permutation	$O(n)^{\uparrow}$	$O(\log n)^*$
Range minimum query	$O(n)$	$O(\log n)$
Tree contraction	$O(n)$	$O(\log n)^*$
Ordered-set operations (Union, intersect, diff)	$O\left(m \log\left(\frac{n}{m}+1\right)\right)$	$O(\log n)$

$^{\uparrow}$ expected, * with high probability

Complexity results

- Let $\mathcal{BF}^1$ be the class of algorithms requiring $O(\log n)$ depth/span and polynomial work in the BF model

$$\mathcal{NC}^1 \subseteq \mathcal{L} \subseteq \mathcal{BF}^1 \subseteq \mathcal{AC}^1 \subseteq \mathcal{PRAM}^1_{CRCW} \subseteq \mathcal{NC}^2$$

- Let $\mathcal{BF}(W(n), S(n))$ be the class of algorithms requiring $S(n)$ depth/span and $W(n)$ work

$$\mathcal{BF}_{\text{Fork-join}} \subsetneq \mathcal{BF} \subseteq \mathcal{BF}_{CAS}$$

↑

[Manuscript from GJS]

Modern parallel machines are highly asynchronous, and global synchronization is costly and should be avoided if possible

1 The binary-forking model and variants

Our suggested model to design and analysis algorithms, and related complexity results

2 Optimal algorithms in the new model

All use no or constant rounds of global synchronization
Many are surprisingly simple

Modern parallel machines are highly asynchronous, and global synchronization is costly and should be avoided if possible

1 The binary-forking model and variants

Our suggested model to design and analysis algorithms, and related complexity results

2 Optimal algorithms in the new model

All use no or constant rounds of global synchronization
Many are surprisingly simple

Future work on exploring asynchrony in parallel algorithms

Complexity, algorithms, implementation, software, etc.

Modern parallel machines are highly asynchronous, and global

Full version of this talk (40 min):

https://www.cs.ucr.edu/~ygu/video/bf.mp4

(or just search "Yan Gu UCR")

Motivations from system and architecture (1:00 – 3:20)

Benchmarking global synchronization cost (4:30 – 6:30)

More discussion for multithreaded models (11:30 – 18:00)

Complexity theory (18:00 – 19:00)

Related work in the BF model (19:00 – 21:20)

List contraction in more details (22:00 – 33:00)

How does this idea generalize to other problems (33:00 – 36:00)

Comparison sort and ordered-set operations (36:40 – 38:00)

Various anecdotes (random places)