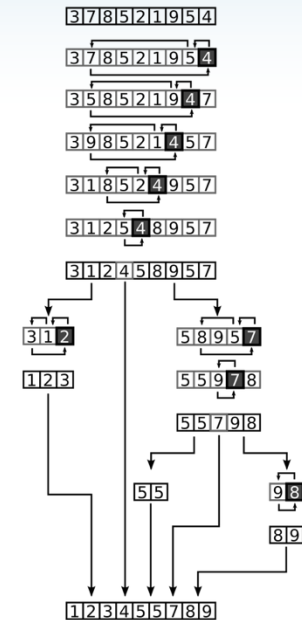


6.506: Algorithm Engineering



LECTURE 2 PARALLEL ALGORITHMS

Julian Shun

September 15, 2026

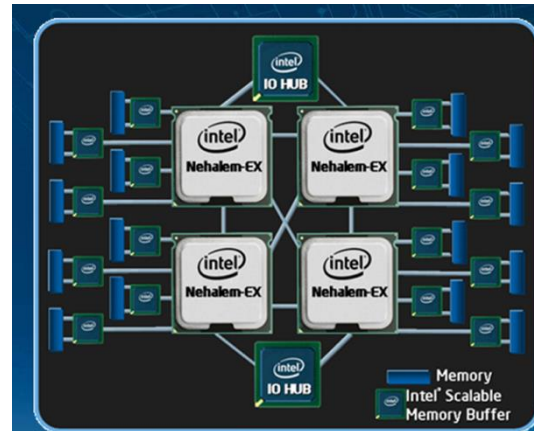
Some lecture material taken from “Parallel Algorithms” by Guy Blelloch and Bruce Maggs.



Announcements

- Presentation sign-up sheet has been posted
- First quiz on Thursday 9/17
 - Covers readings from 9/10, 9/15, and 9/17

Parallel Computing

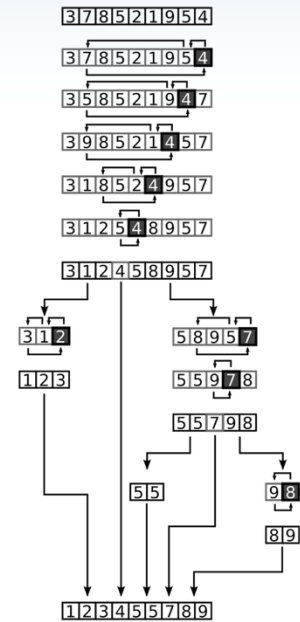


Parallel Languages

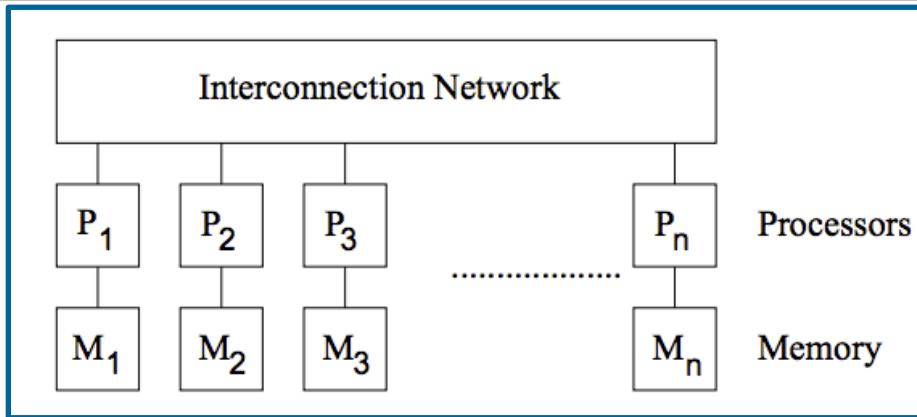
- Pthreads
- Cilk, OpenMP
- Message Passing Interface (MPI)
- CUDA, OpenCL
- Today: Shared-memory parallelism
 - Cilk and OpenMP are extensions of C/C++ that support parallel for-loops, parallel recursive calls, etc.
 - Do not need to worry about assigning tasks to processors as these languages have a runtime scheduler
 - Cilk has a provably efficient runtime scheduler



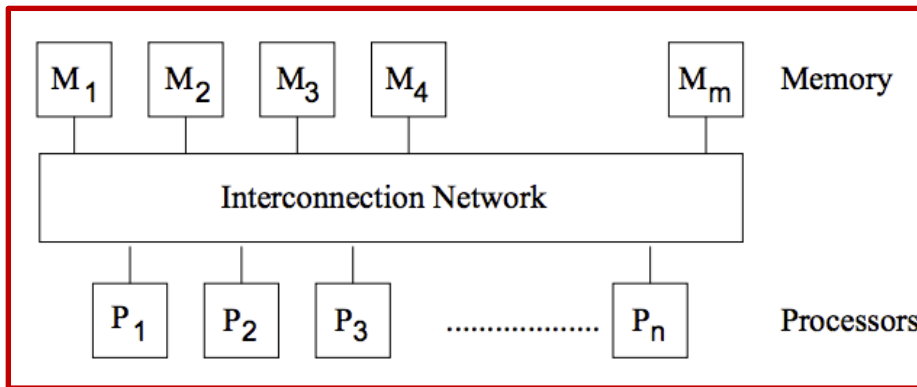
PARALLELISM MODELS



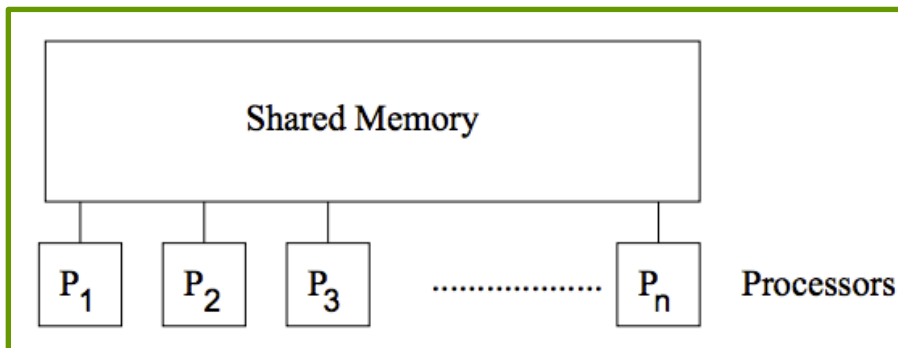
Basic multiprocessor models



Local memory machine



Modular memory machine

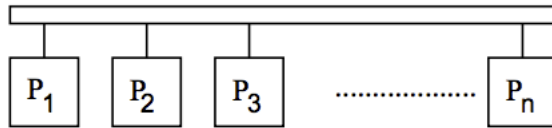


Parallel random-access Machine (PRAM)

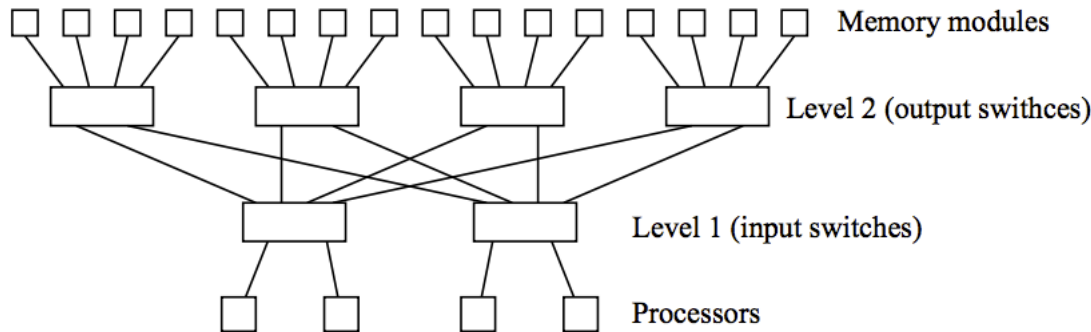
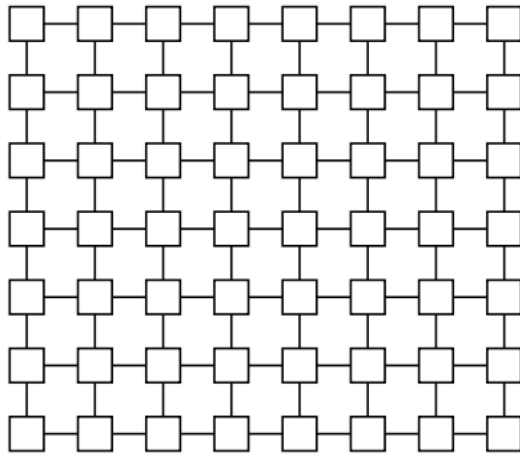
Source: "Parallel Algorithms" by Guy E. Blelloch and Bruce M. Maggs

Network topology

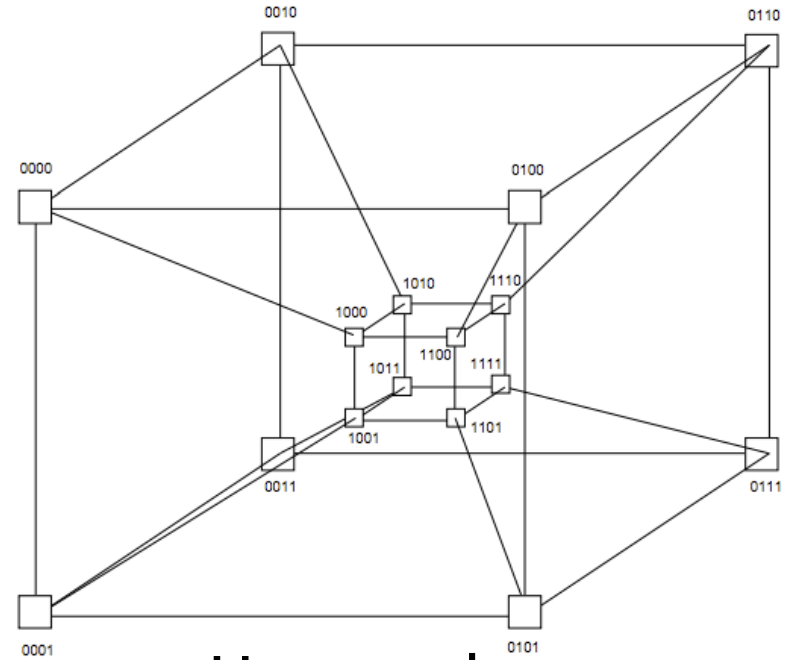
Bus



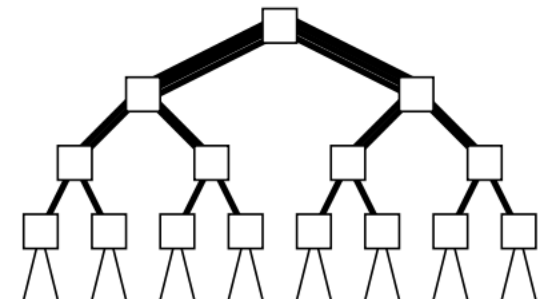
Mesh



2-level multistage network



Hypercube



Fat tree

Source: "Parallel Algorithms" by Guy E. Blelloch and Bruce M. Maggs

Network topology

- Algorithms for specific topologies can be complicated
 - May not perform well on other networks
- Alternative: use a model that summarizes latency and bandwidth of network
 - Postal model
 - Bulk-Synchronous Parallel (BSP) model
 - LogP model

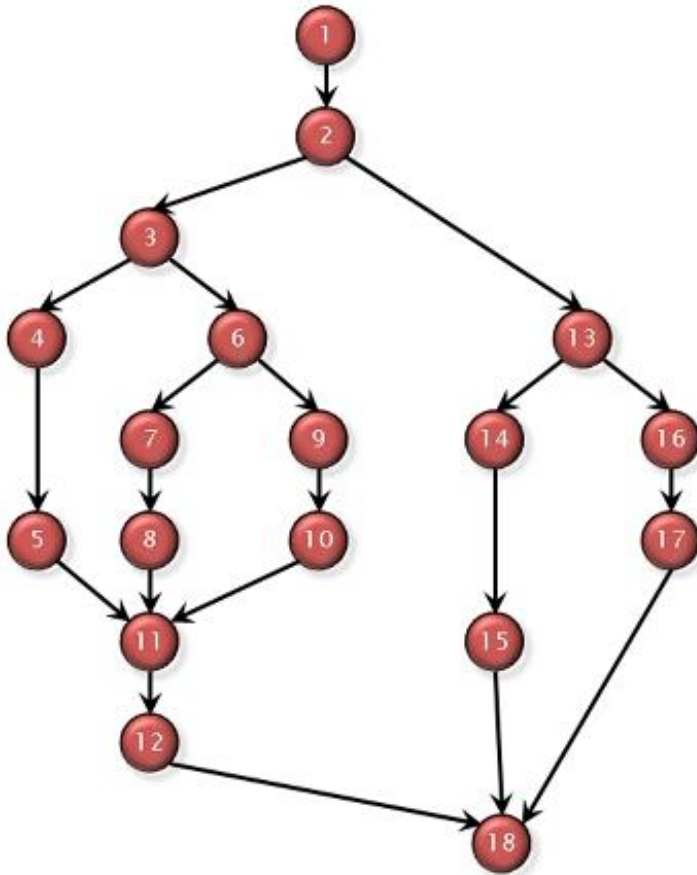
PRAM Model

- All processors can perform same local instructions as in the RAM model
- All processors operate in lock-step
- Implicit synchronization between steps
- Models for concurrent access
 - Exclusive-read exclusive-write (EREW)
 - Concurrent-read concurrent-write (CRCW)
 - How to resolve concurrent writes: arbitrary value, value from lowest-ID processor, logical OR of values, sum of values
 - Concurrent-read exclusive-write (CREW)
 - Queue-read queue-write (QRQW)
 - Allows concurrent access in time proportional to the maximal number of concurrent accesses

Work-Span model

- Similar to PRAM but does not require lock-step or processor allocation

Computation graph



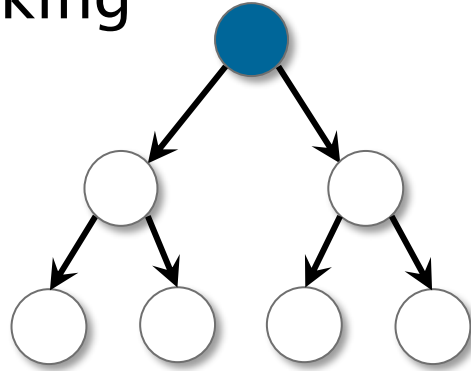
- Work** = number of vertices in graph (number of operations)
- Span (Depth)** = longest directed path in graph (dependence length)
- Parallelism** = $\text{Work} / \text{Span}$
- A **work-efficient** parallel algorithm has work that asymptotically matches the best sequential algorithm for the problem

Goal: work-efficient and low (polylogarithmic) span parallel algorithms

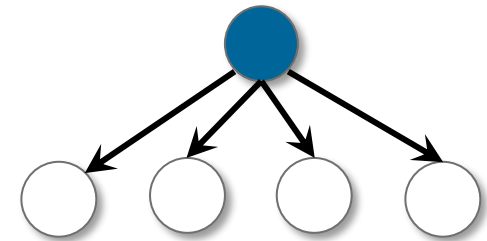
Work-Span model

- Spawning/forking tasks

- Model can support either binary forking or arbitrary forking



Binary forking



Arbitrary forking

- Cilk uses binary forking, as seen in 6.106/6.172
- Converting between the two changes work by at most a constant factor and span by at most a logarithmic factor
 - Keep this in mind when reading textbooks/papers on parallel algorithms
- We will assume arbitrary forking unless specified

Work-Span model

- State what operations are supported
 - Concurrent reads/writes?
 - Resolving concurrent writes

Scheduling

- For a computation with work W and span S , on P processors a greedy scheduler achieves

$$\text{Running time} \leq W/P + S$$

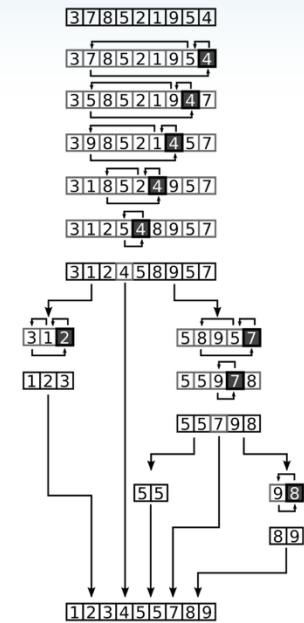
- For a computation with work W and span S , on P processors Cilk's work-stealing scheduler achieves

$$\text{Expected running time} \leq W/P + O(S)$$

- Work-efficiency is important since P and S are usually small



PARALLEL SUM



Parallel Sum

- Definition: Given a sequence $A=[x_0, x_1, \dots, x_{n-1}]$, return $x_0 + x_1 + \dots + x_{n-2} + x_{n-1}$

What is the span?

$$S(n) = S(n/2) + O(1)$$

$$S(1) = O(1)$$

$$\rightarrow S(n) = O(\log n)$$

What is the work?

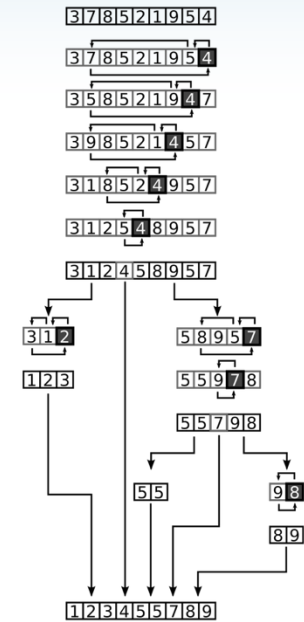
$$W(n) = W(n/2) + O(n)$$

$$W(1) = O(1)$$

$$\rightarrow W(n) = O(n)$$



PREFIX SUM



Prefix Sum

- Definition: Given a sequence $A=[x_0, x_1, \dots, x_{n-1}]$, return a sequence where each location stores the sum of everything before it in A , $[0, x_0, x_0+x_1, \dots, x_0+x_1+\dots+x_{n-2}]$, as well as the total sum $x_0+x_1+\dots+x_{n-2}+x_{n-1}$

- Example:

2	4	3	1	3
---	---	---	---	---



0	2	6	9	10
---	---	---	---	----

Total sum = 13

- Can be generalized to any associative binary operator (e.g., $\times$, min, max)

Sequential Prefix Sum

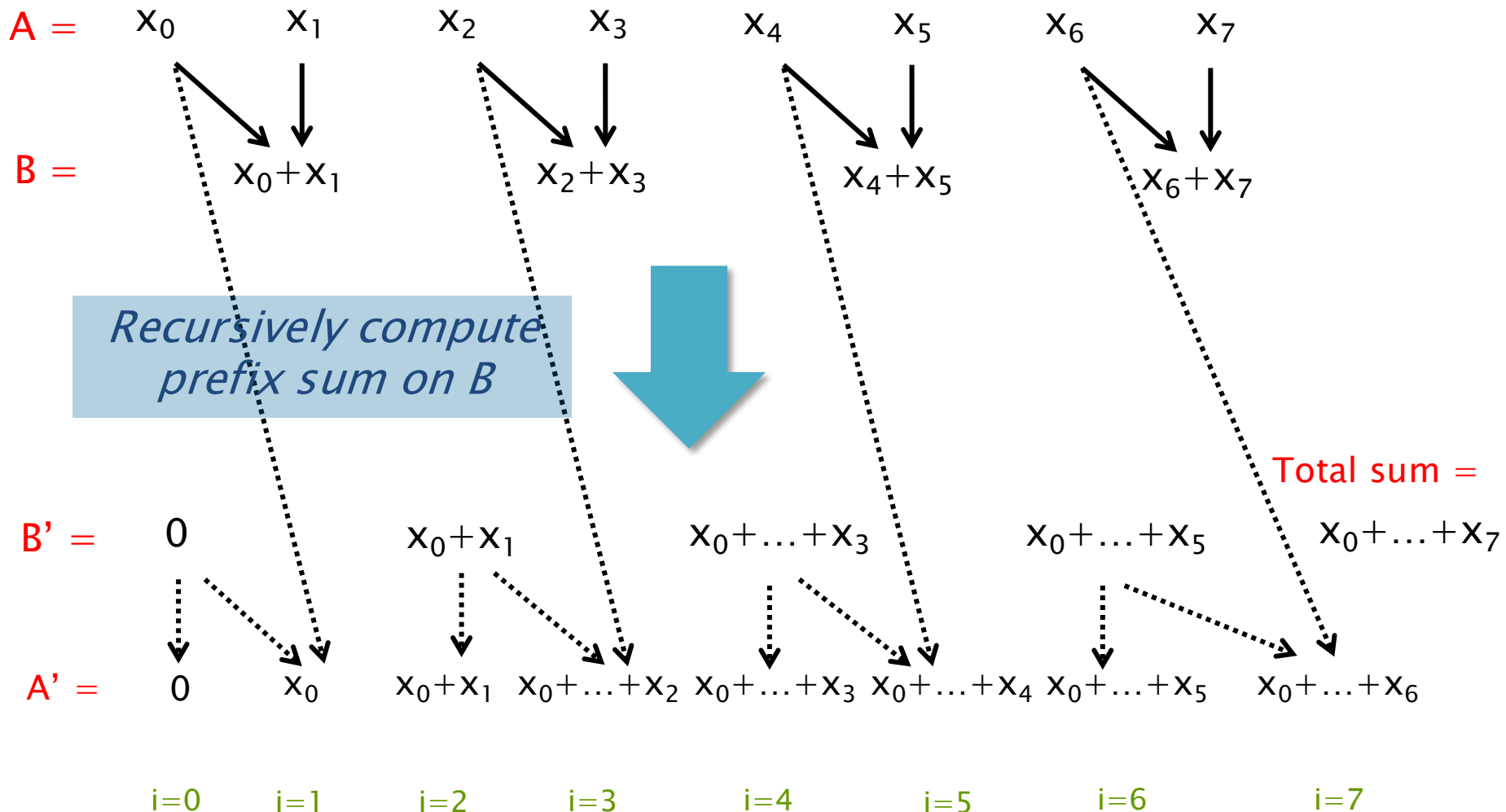
Input: array A of length n

Output: array A' and total sum

```
cumulativeSum = 0;  
for i=0 to n-1:  
     $A'[i] = \text{cumulativeSum}$ ;  
     $\text{cumulativeSum} += A[i]$ ;  
return  $A'$  and cumulativeSum
```

- What is the work of this algorithm?
 - $O(n)$
- Can we execute iterations in parallel?
 - Loop carried dependence: value of cumulativeSum depends on previous iterations

Parallel Prefix Sum



Total sum =

$x_0+\dots+x_7$

Total sum =

$x_0+\dots+x_7$

for even values of i : $A'[i] = B'[i/2]$

for odd values of i : $A'[i] = B'[(i-1)/2] + A[i-1]$

Parallel Prefix Sum

Input: array A of length n (assume n is a power of 2)

Output: array A' and total sum

PrefixSum(A, n):

if $n == 1$: return ([0], A[0])

for $i=0$ to $n/2-1$ in parallel:

$B[i] = A[2i] + A[2i+1]$

$(B', \text{sum}) = \text{PrefixSum}(B, n/2)$

for $i=0$ to $n-1$ in parallel:

if $(i \bmod 2) == 0$: $A'[i] = B'[i/2]$

else: $A'[i] = B'[(i-1)/2] + A[i-1]$

return (A', sum)

What is the span?

$$S(n) = S(n/2) + O(1)$$

$$S(1) = O(1)$$

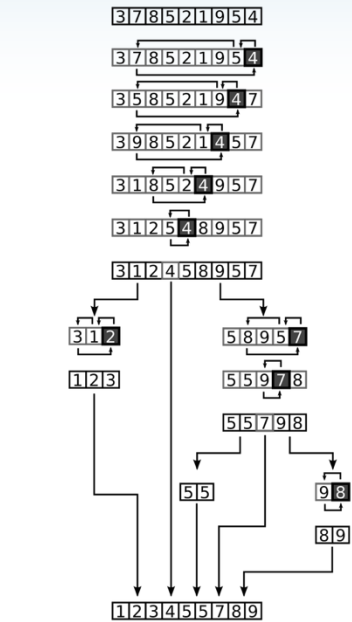
$$\rightarrow S(n) = O(\log n)$$

What is the work?

$$W(n) = W(n/2) + O(n)$$

$$W(1) = O(1)$$

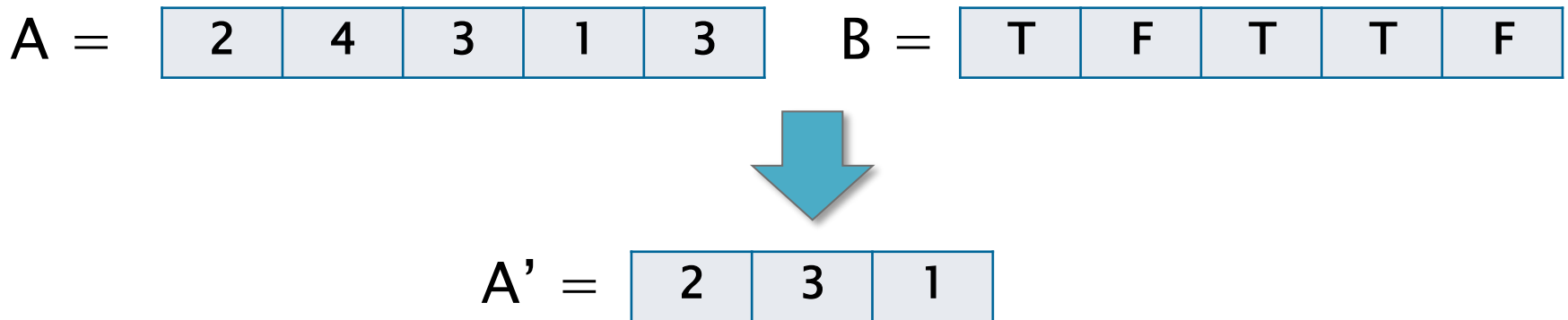
$$\rightarrow W(n) = O(n)$$



A stylized illustration of a desktop computer system. It features a large monitor with a blue screen, a black tower unit, and a black keyboard. The components are arranged on a white background.

Filter

- Definition: Given a sequence $A=[x_0, x_1, \dots, x_{n-1}]$ and a Boolean array of flags $B[b_0, b_1, \dots, b_{n-1}]$, output an array A' containing just the elements $A[i]$ where $B[i] = \text{true}$ (maintaining relative order)
- Example:



- Can you implement filter using prefix sum?

Filter Implementation

A =

2	4	3	1	3
---	---	---	---	---

B =

T	F	T	T	F
---	---	---	---	---

1	0	1	1	0
---	---	---	---	---

```
// Assume B'[n] = total sum
parallel-for i=0 to n-1:
  if(B'[i] != B'[i+1]):
    A'[B'[i]] = A[i];
```



Prefix sum

B' =

0	1	1	2	3
---	---	---	---	---

Total sum = 3

Allocate array A' of size 3

--	--	--

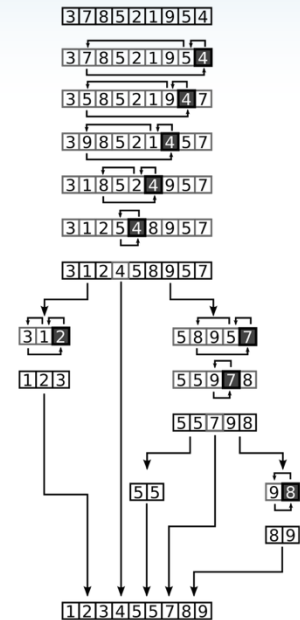


A' =

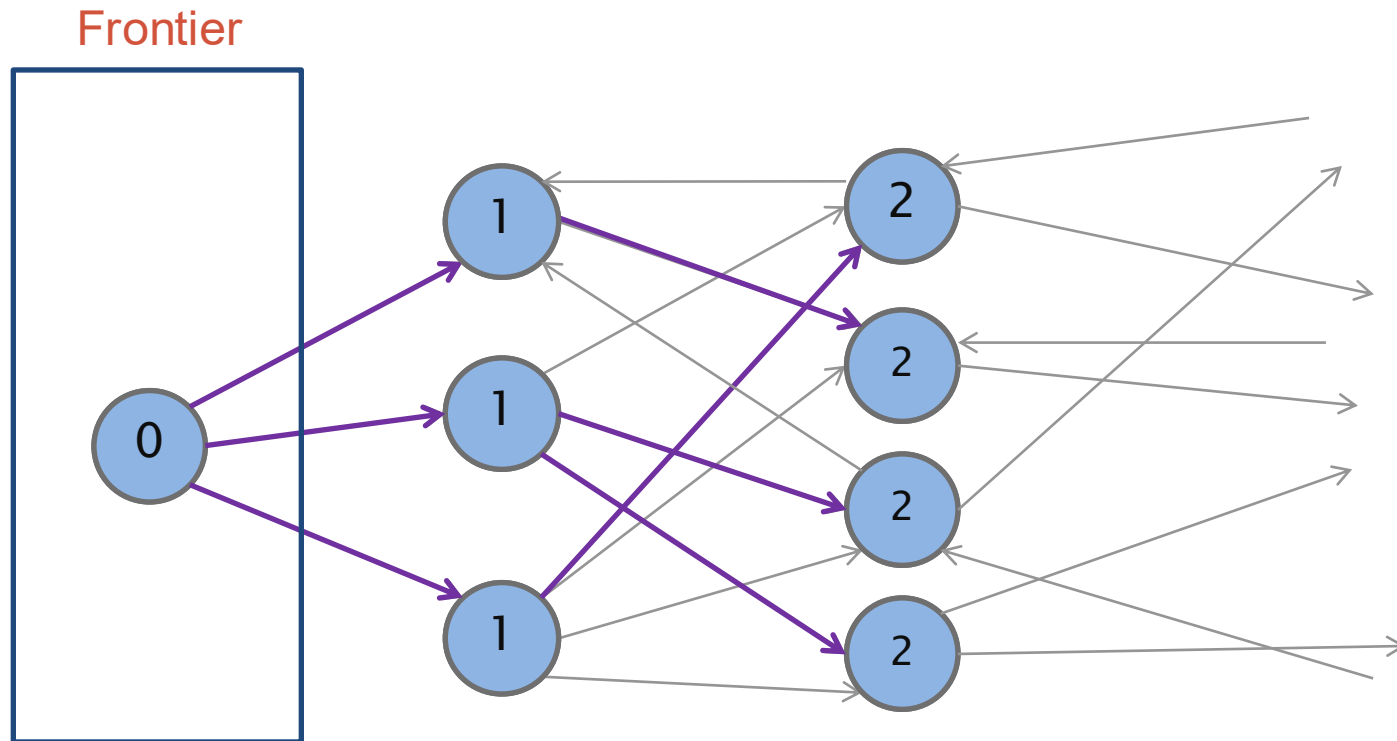
2	3	1
---	---	---



PARALLEL BREADTH-FIRST SEARCH



Parallel BFS Algorithm



- Can process each frontier in parallel
 - Parallelize over both the vertices and their outgoing edges

Parallel BFS Code

```

BFS(Offsets, Edges, source) {
    parent, frontier, frontierNext, and degrees are array
    parallel_for(int i=0; i<n; i++) parent[i] = -1;
    frontier[0] = source, frontierSize = 1, parent[source] = source;

    while(frontierSize > 0) {
        parallel_for(int i=0; i<frontierSize; i++)
            degrees[i] = Offsets[frontier[i]+1] - Offsets[frontier[i]];
        perform prefix sum on degrees array
        parallel_for(int i=0; i<frontierSize; i++) {
            v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
            for(int j=0; j<d; j++) { //can be parallel
                ngh = Edges[Offsets[v]+j];
                if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
                    frontierNext[index+j] = ngh;
                } else { frontierNext[index+j] = -1; }
            }
        }
        filter out "-1" from frontierNext, store in frontier, and update frontierSize to be
        the size of frontier (all done using prefix sum)
    }
}

```

frontierSize = 5

2	4	3	1	3
---	---	---	---	---

Prefix sum



0	2	6	9	10
---	---	---	---	----

frontierSize = 8

24	9	24	9	15	89	25	90	99	4	-1	1	4	8
----	---	----	---	----	----	----	----	----	---	----	---	---	---

frontierSize = 8

Diagram illustrating the parallel BFS process. The frontier array is updated from [2, 4, 3, 1, 3] to [24, 9, 24, 9, 15, 89, 25, 90, 99, 4, -1, 1, 4, 8]. The frontierSize is updated from 5 to 8. The diagram shows the frontier array and the degrees array (2, 4, 3, 1, 3) and its prefix sum (0, 2, 6, 9, 10). The frontier array is divided into segments by curly braces, with nodes v1, v2, v3, v4, and v5 labeled above them. The final frontier array is shown with the last element -1, and the frontierSize is updated to 8.

BFS Work-Span Analysis

- Number of iterations $\leq$ diameter Δ of graph
- Each iteration takes $O(\log m)$ span for prefix sum and filter (assuming inner loop is parallelized)

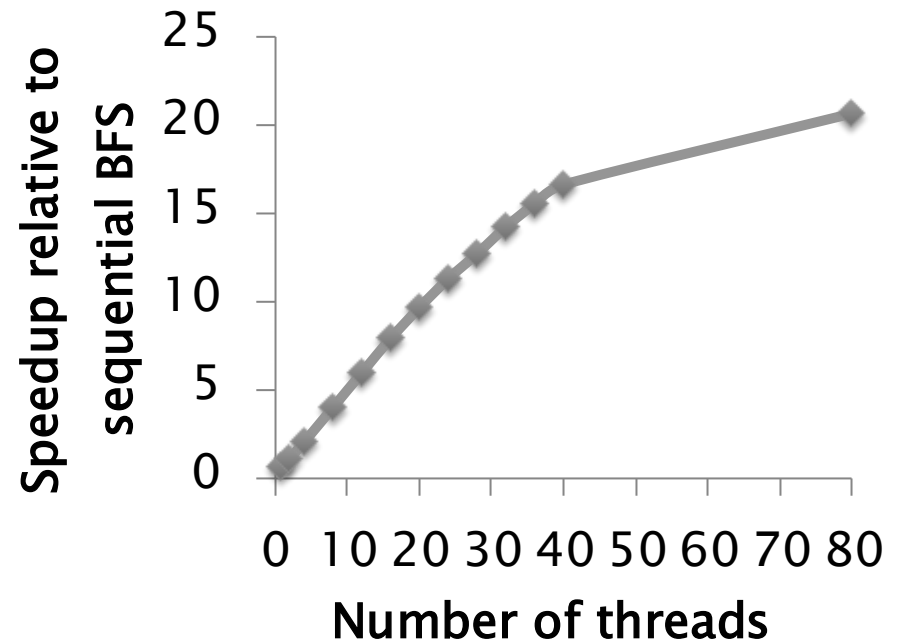
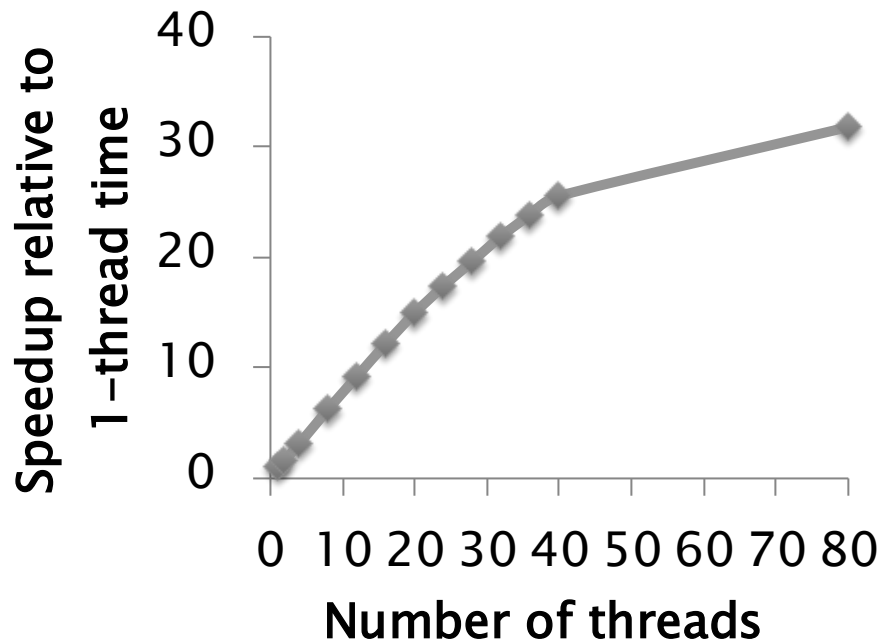
$$\text{Span} = O(\Delta \log m)$$

- Sum of frontier sizes = n
- Each edge traversed once $\rightarrow m$ total visits
- Work of prefix sum on each iteration is proportional to frontier size $\rightarrow O(n)$ total
- Work of filter on each iteration is proportional to number of edges traversed $\rightarrow O(m)$ total

$$\text{Work} = O(n+m)$$

Performance of Parallel BFS

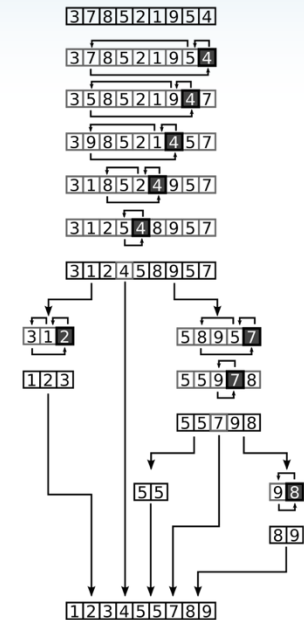
- Random graph with $n=10^7$ and $m=10^8$
 - 10 edges per vertex
- 40-core machine with 2-way hyperthreading



- 31.8x speedup on 40 cores with hyperthreading
- Sequential BFS is 54% faster than parallel BFS on 1 thread

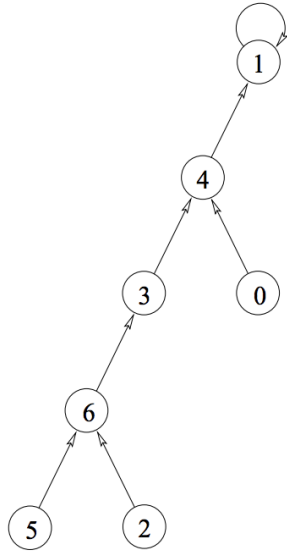


POINTER JUMPING AND LIST RANKING

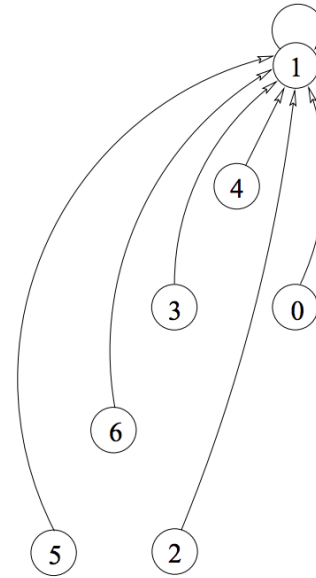


Pointer Jumping

- Have every node in linked list or rooted tree point to the end (root)



(a) The input tree $P = [4, 1, 6, 4, 1, 6, 3]$.



(b) (c) The final tree $P = [1, 1, 1, 1, 1, 1, 1]$. iteration

```
for j=0 to ceil(log n)-1:
  parallel-for i=0 to n-1:
    temp[i] = P[P[i]];
  parallel-for i=0 to n-1:
    P[i] = temp[i];
```

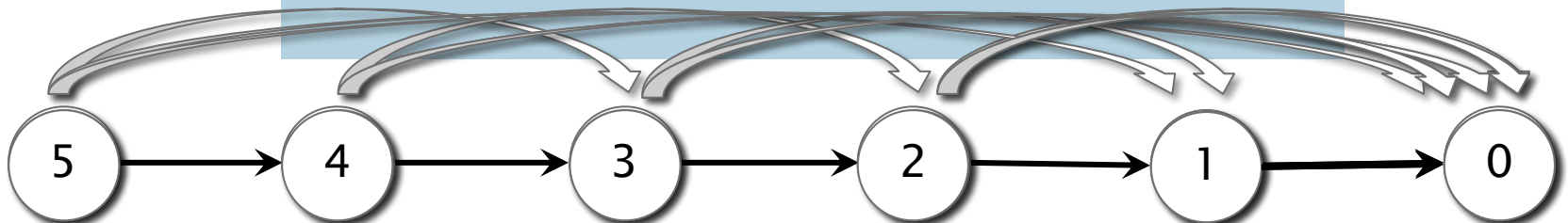
What is the work and span?

$$W = O(n \log n)$$
$$S = O(\log n)$$

List Ranking

- Have every node in linked list determine its distance to the end

```
parallel-for i=0 to n-1:  
  if P[i] == i then rank[i] = 0  
  else rank[i] = 1  
  
for j=0 to ceil(log n)-1:  
  temp, temp2;  
  parallel-for i=0 to n-1:  
    temp[i] = rank[P[i]];  
    temp2[i] = P[P[i]];  
  parallel-for i=0 to n-1:  
    rank[i] = rank[i] + temp[i];  
    P[i] = temp2[i];
```



Work-Span Analysis

```
parallel-for i=0 to n-1:  
  if P[i] == i then rank[i] = 0  
  else rank[i] = 1  
  
for j=0 to ceil(log n)-1:  
  temp, temp2;  
  parallel-for i=0 to n-1:  
    temp = rank[P[i]];  
    temp2 = P[P[i]];  
  parallel-for i=0 to n-1:  
    rank[i] = rank[i] + temp;  
    P[i] = temp2;
```

What is the work and span?

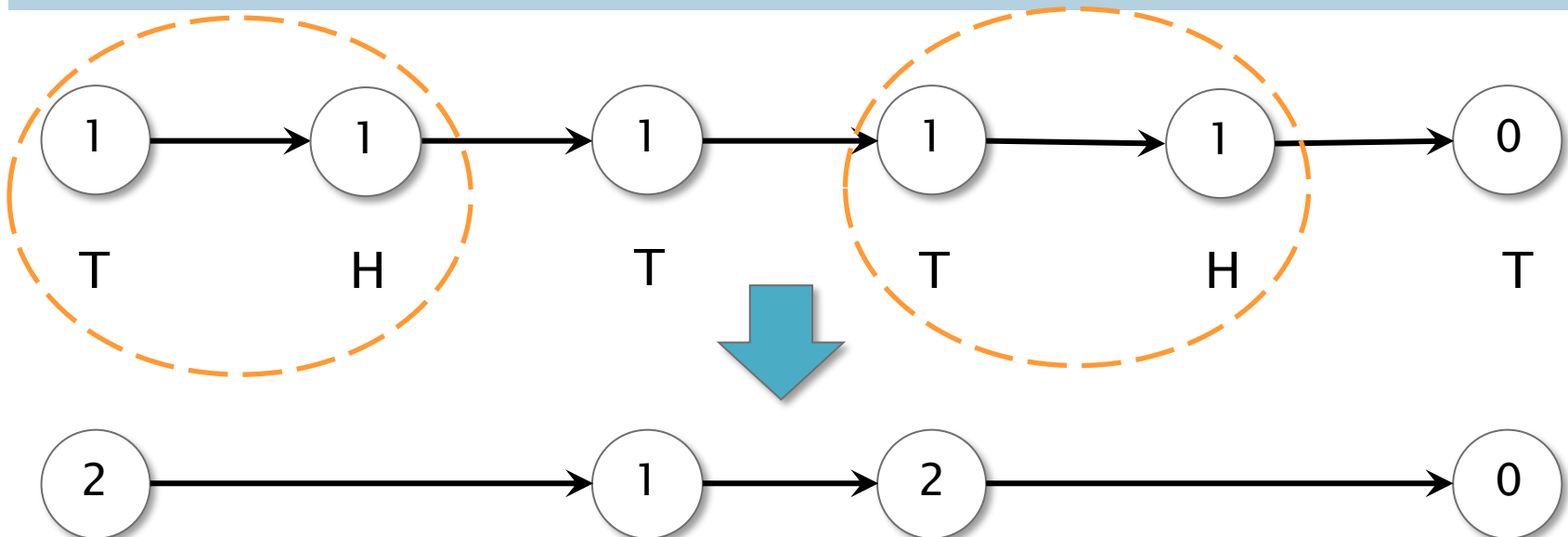
$$W = O(n \log n)$$
$$S = O(\log n)$$

Sequential algorithm only requires $O(n)$ work

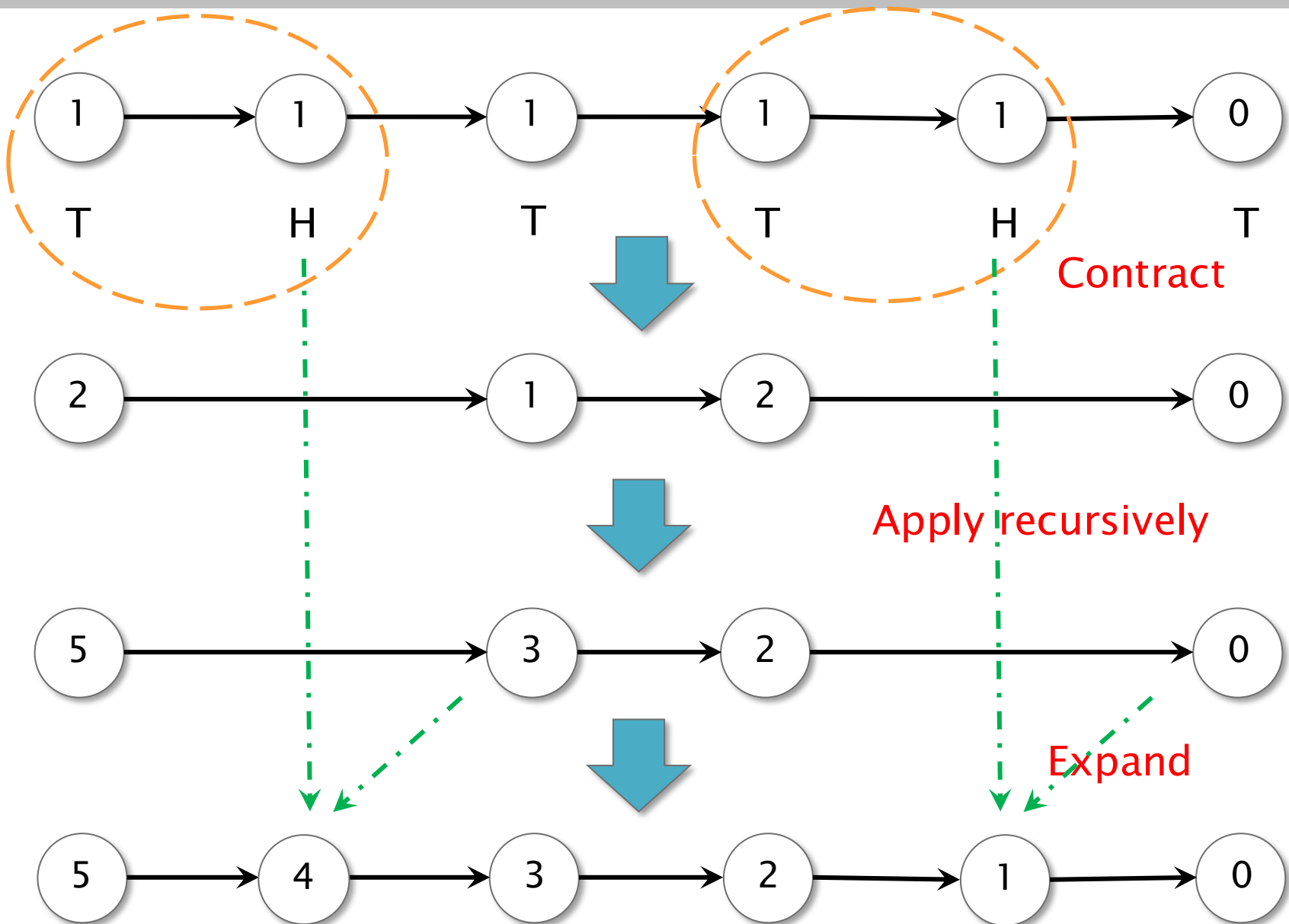
Work-Efficient List Ranking

ListRanking(list P)

1. If list has two or fewer nodes, then return **//base case**
2. Every node flips a fair coin
3. For each vertex u (except the last vertex), if u flipped Tails and $P[u]$ flipped Heads then u will be paired with $P[u]$
 - A. $\text{rank}[u] = \text{rank}[u] + \text{rank}[P[u]]$
 - B. $P[u] = P[P[u]]$
4. Recursively call ListRanking on smaller list
5. Insert contracted nodes v back into list with $\text{rank}[v] = \text{rank}[v] + \text{rank}[P[v]]$



Work-Efficient List Ranking

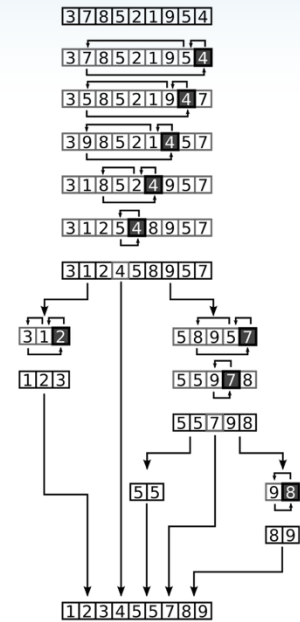


Work-Span Analysis

- Number of pairs per round is $(n-1)/4$ in expectation
 - For all nodes u except for the last node, probability of u flipping Tails and $P[u]$ flipping Heads is $1/4$
 - Linearity of expectations gives $(n-1)/4$ pairs overall
- Each round takes linear work and $O(1)$ span
- Expected work: $W(n) \leq W(7n/8) + O(n)$
- Expected span: $S(n) \leq S(7n/8) + O(1)$

$$W = O(n)$$
$$S = O(\log n)$$

- Can show span with high probability with Chernoff bound



CONNECTED COMPONENTS

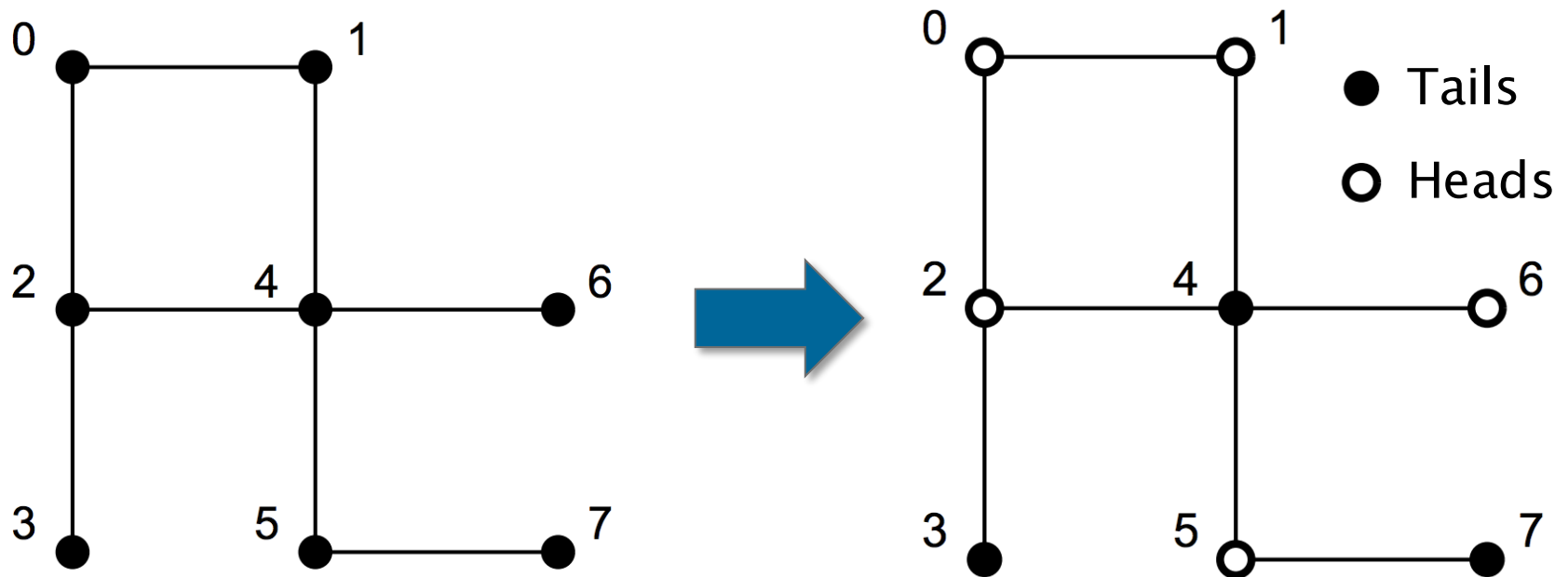


Connected Components

- Given an undirected graph, label all vertices such that $L(u) = L(v)$ if and only if there is a path between u and v
- BFS span is proportional to diameter
 - Works well for graphs with small diameter
- Today we will see a randomized algorithm that takes $O((n+m)\log n)$ work and $O(\log n)$ span
 - Deterministic version in paper
 - Later this semester, we will study a work-efficient parallel algorithm

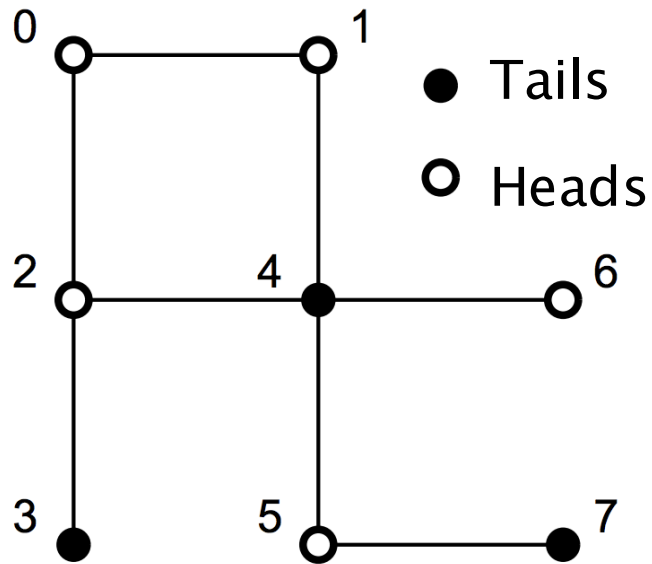
Random Mate

- Idea: Form a set of non-overlapping star subgraphs and contract them
- Each vertex flips a coin. For each Heads vertex, pick an arbitrary Tails neighbor (if there is one) and point to it

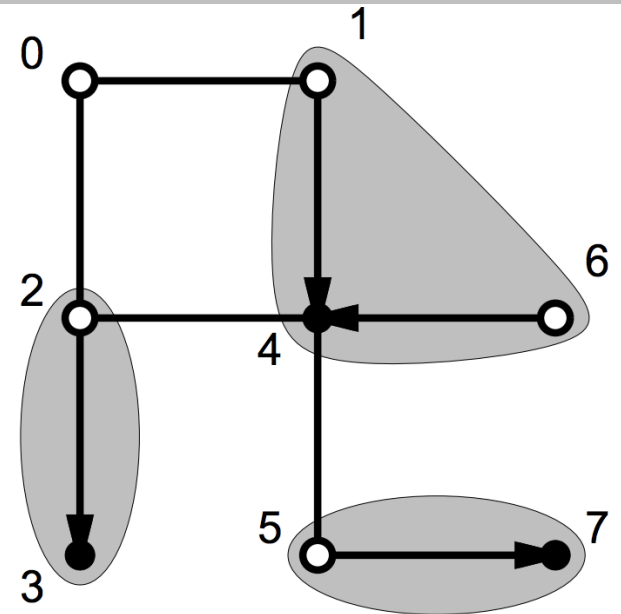
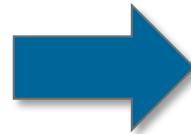


Source: "Parallel Algorithms" by Guy E. Blelloch and Bruce M. Maggs

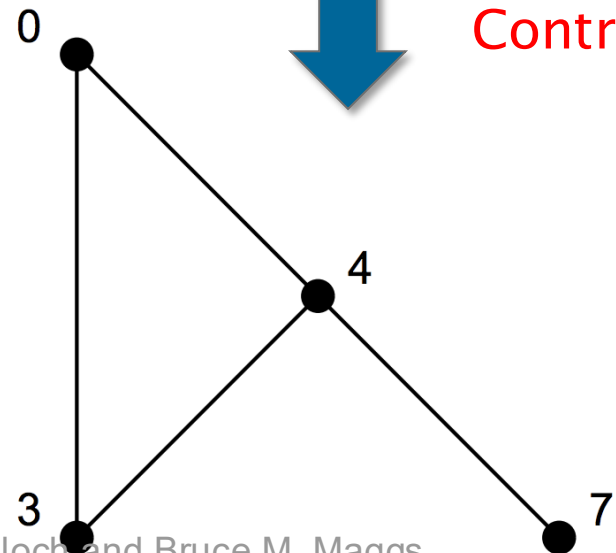
Random Mate



Form stars



Contract



Repeat until each component has a single vertex

Expand vertices back in reverse order with label of neighbor

Source: "Parallel Algorithms" by Guy E. Blelloch and Bruce M. Maggs

Random Mate Algorithm

CC_Random_Mate(L, E)

if($|E| = 0$) Return L //base case

else

1. Flip coins for all vertices
2. For v where $\text{coin}(v)=\text{Heads}$, hook to arbitrary Tails neighbor w and set $L(v) = w$
3. $E' = \{ (L(u), L(v)) \mid (u, v) \in E \text{ and } L(u) \neq L(v) \}$
4. $L' = \text{CC_Random_Mate}(L, E')$
5. For v where $\text{coin}(v)=\text{Heads}$, set $L'(v) = L'(w)$ where w is the Tails neighbor that v hooked to in Step 2
6. Return L'

- Each iteration requires $O(m+n)$ work and $O(1)$ span
 - Assumes we do not pack vertices and edges
- Each iteration eliminates $1/4$ of the vertices in expectation

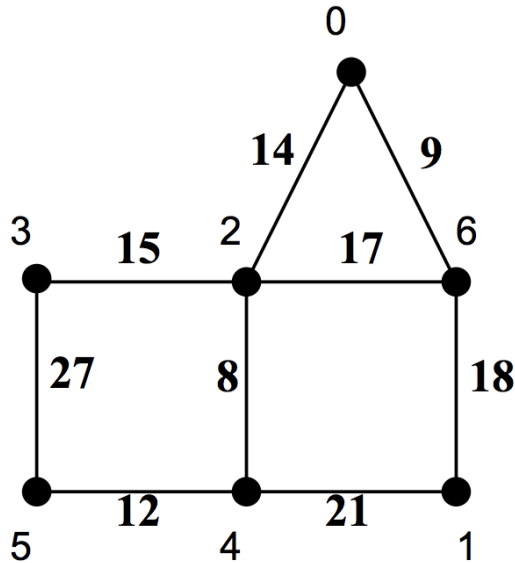
$W = O((m+n)\log n)$ w.h.p.

$S = O(\log n)$ w.h.p.

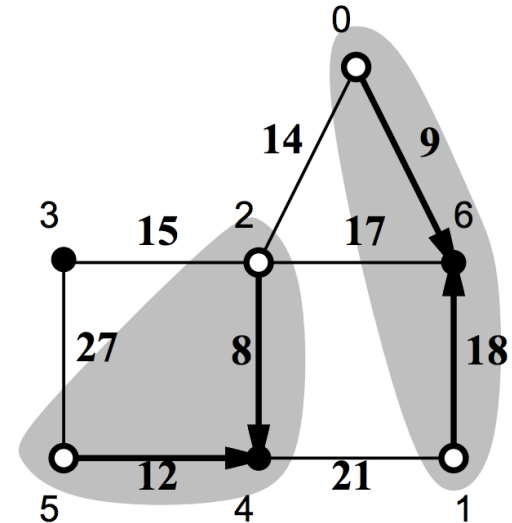
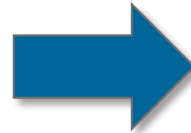
(Minimum) Spanning Forest

- Spanning Forest: Keep track of edges used for hooking
 - Edges will only hook two components that are not yet connected
- Minimum Spanning Forest:
 - For each “Heads” vertex v , instead of picking an arbitrary neighbor to hook to, pick neighbor w where (v, w) is the minimum weight edge incident to v
 - Can find this edge using priority concurrent write

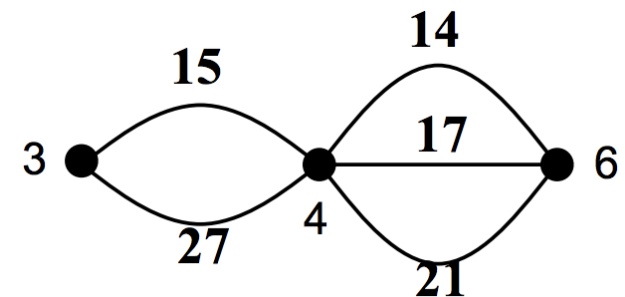
Minimum Spanning Forest



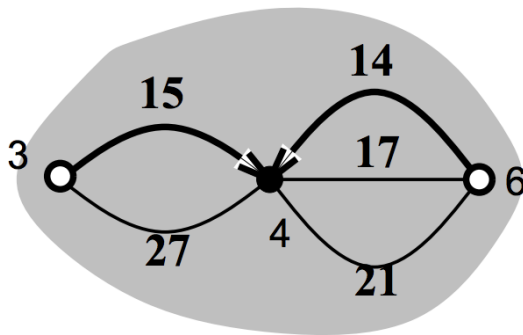
Form stars with
min-weight edge



Contract



Repeat



Source: "Parallel Algorithms" by Guy E. Blelloch and Bruce M. Maggs