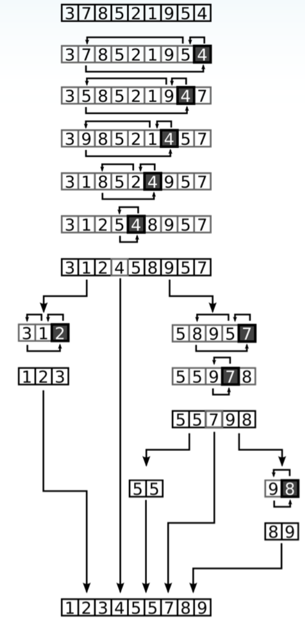
6.886: Algorithm Engineering



LECTURE 2
PARALLEL ALGORITHMS

Julian Shun

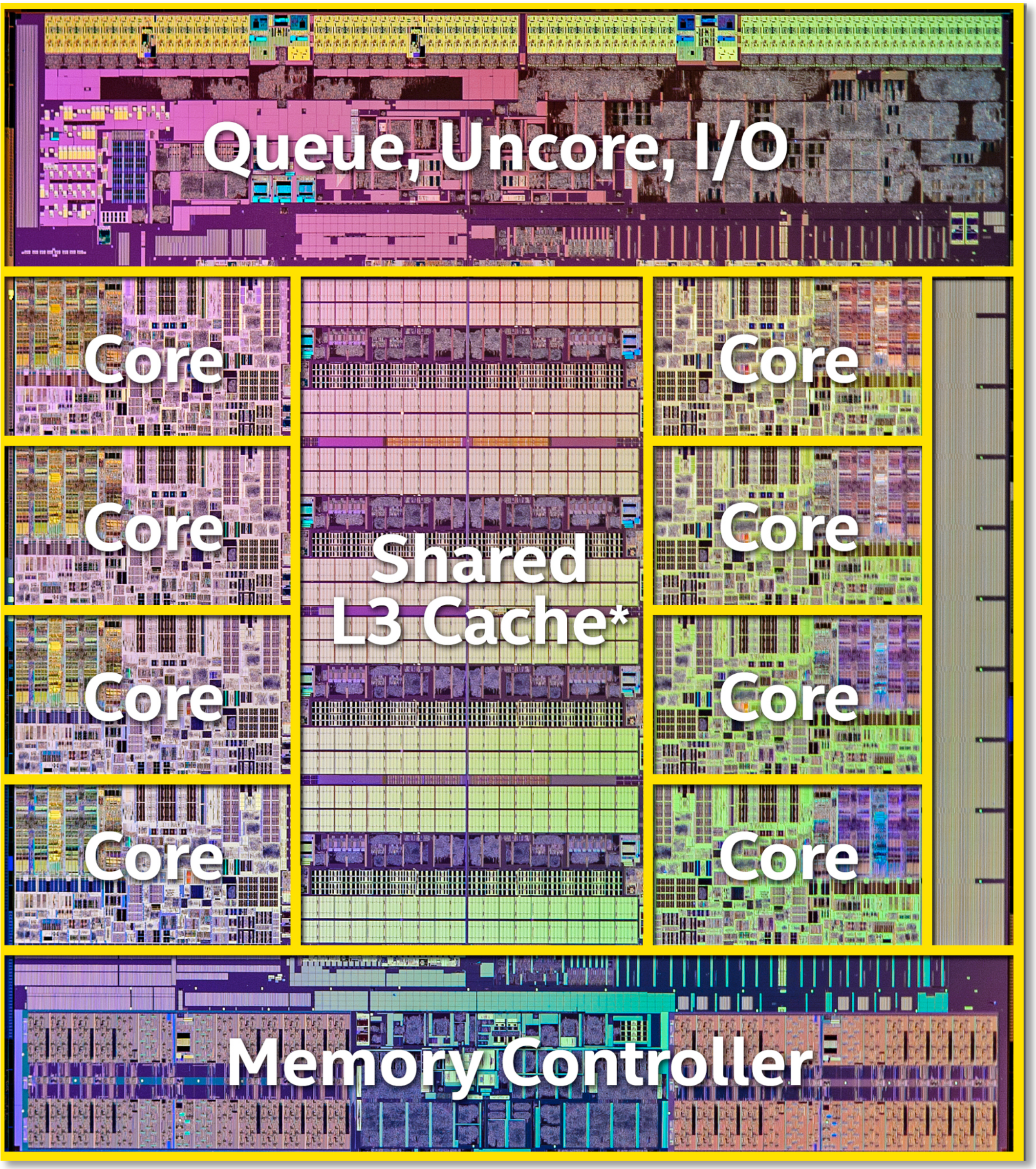
February 18, 2020

Lecture material taken from “Parallel Algorithms” by Guy E. Blelloch and Bruce M. Maggs and 6.172 by Charles Leiserson and Saman Amarasinghe

Announcement

- Presentation sign-up sheet posted on Piazza

Multicore Processors

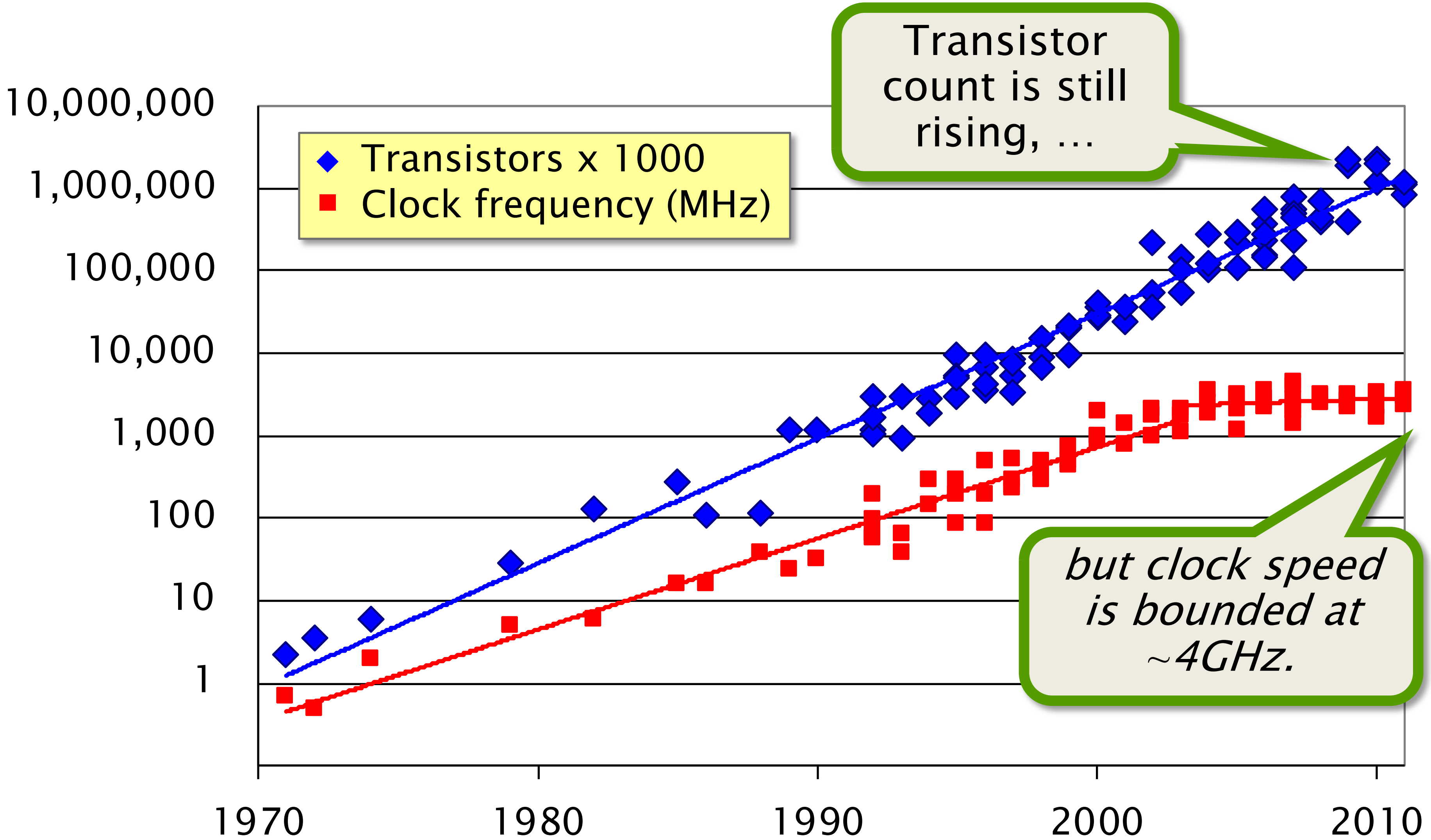


Q Why do semicon-ductor vendors provide chips with multiple processor cores?

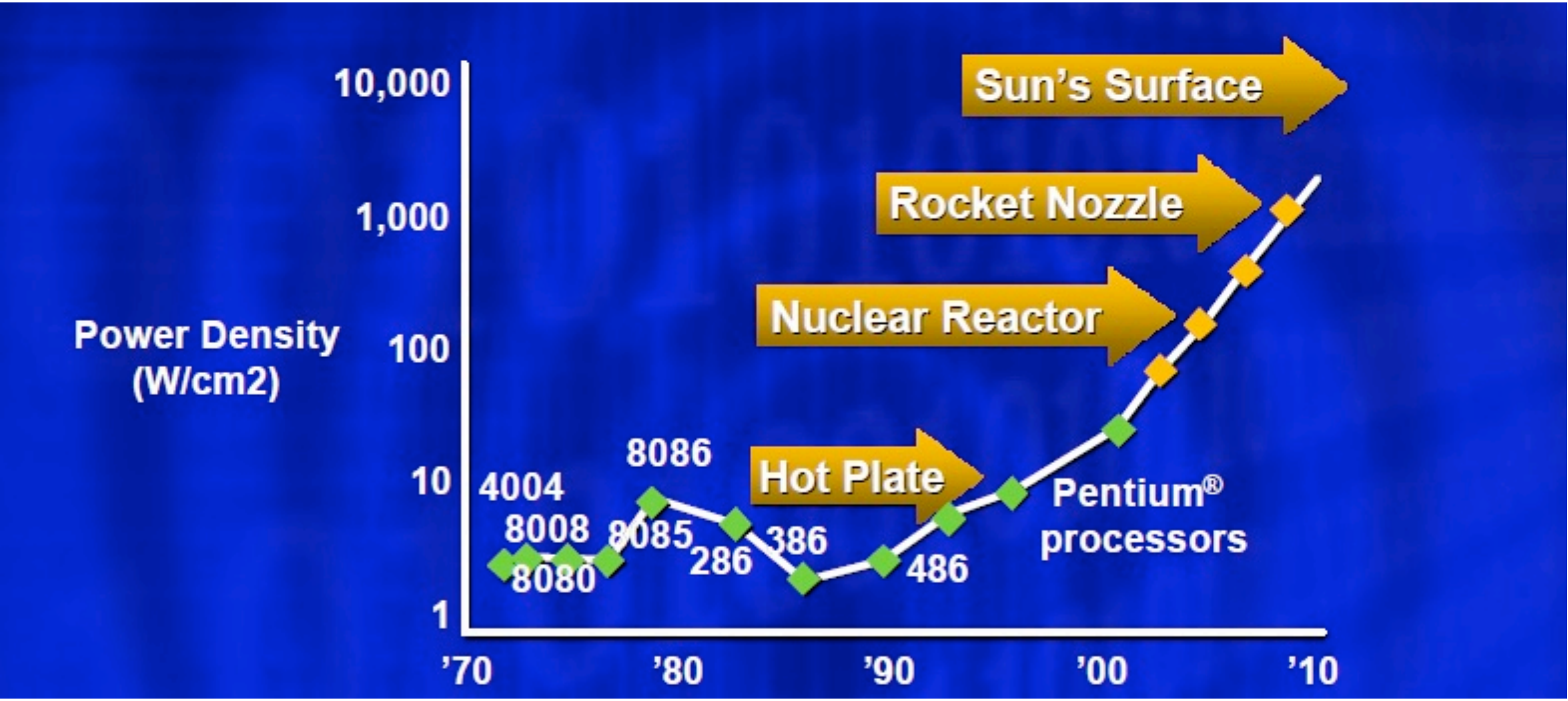
A Because of Moore's Law and the end of the scaling of clock frequency.

Intel Haswell-E

Technology Scaling



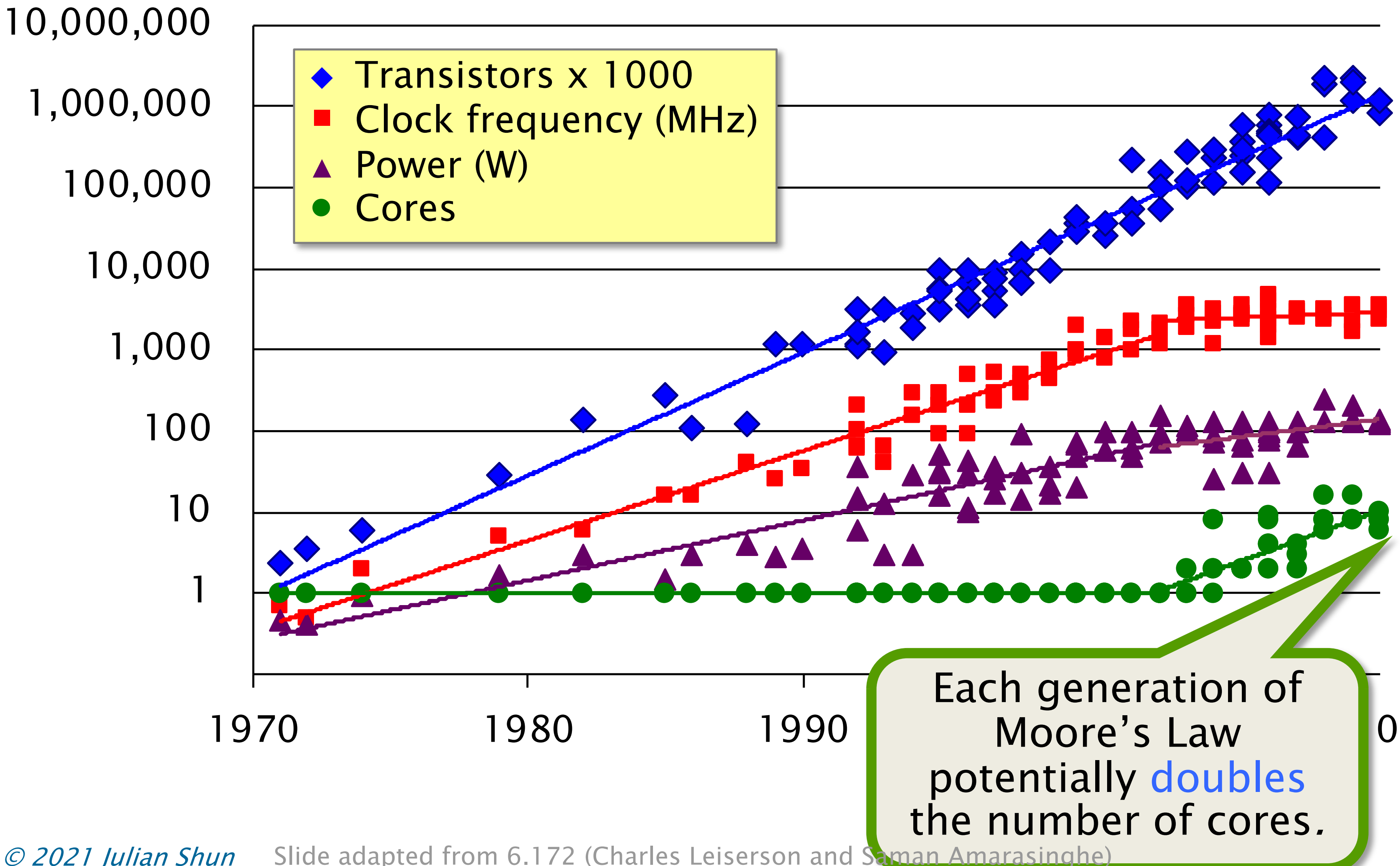
Power Density



Source: Patrick Gelsinger, *Intel Developer's Forum*, Intel Corporation, 2004.

Projected power density, if clock frequency had continued its trend of scaling 25%–30% per year.

Technology Scaling

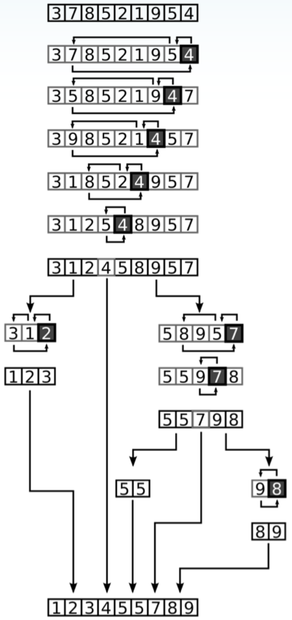


Parallel Languages

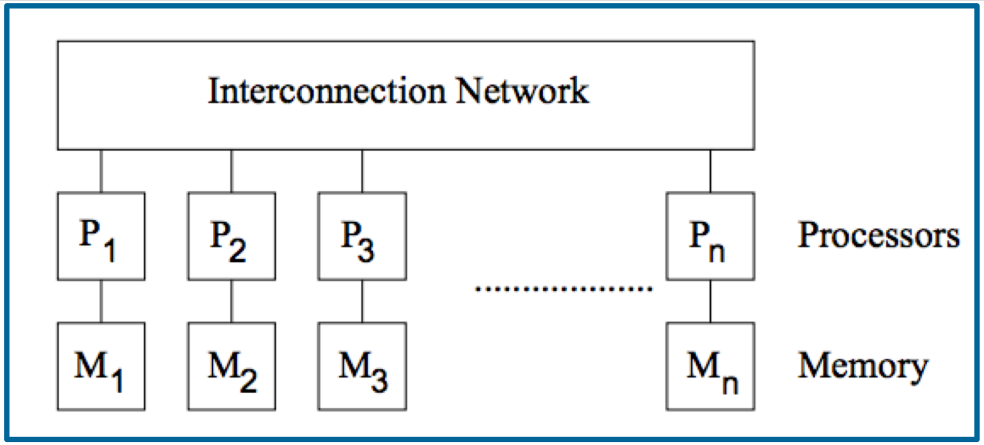
- Pthreads
- Cilk, OpenMP
- Message Passing Interface (MPI)
- CUDA, OpenCL

- Today: Shared-memory parallelism
  - Cilk and OpenMP are extensions of C/C++ that supports 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

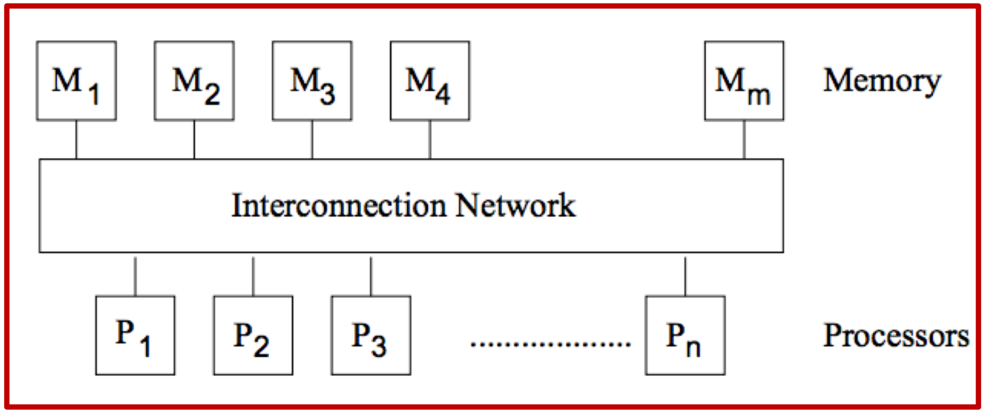
PARALLELISM MODELS



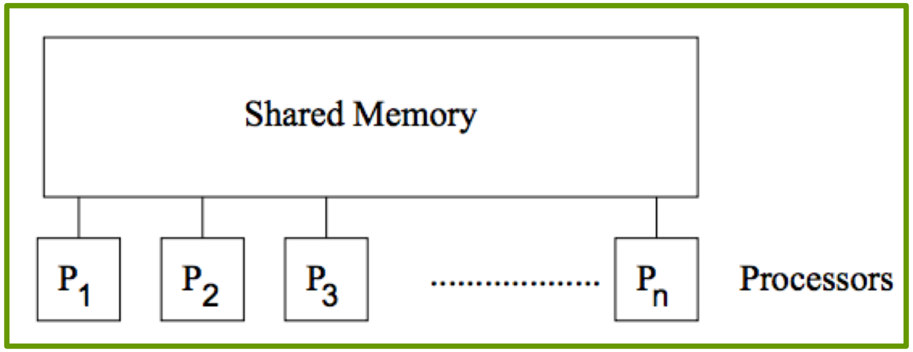
Basic multiprocessor models



Local memory machine



Modular memory machine

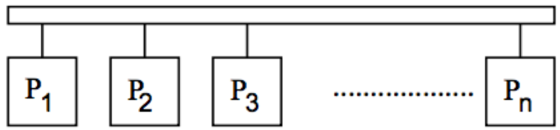


Parallel random-access Machine (PRAM)

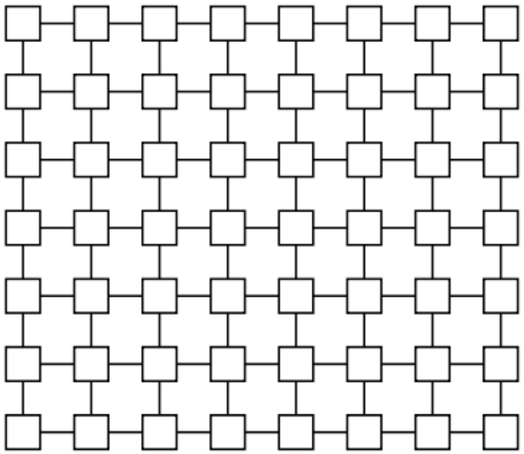
Source: "Parallel Algorithms" by Guy E. Blelloch and Bruce M. Maggs

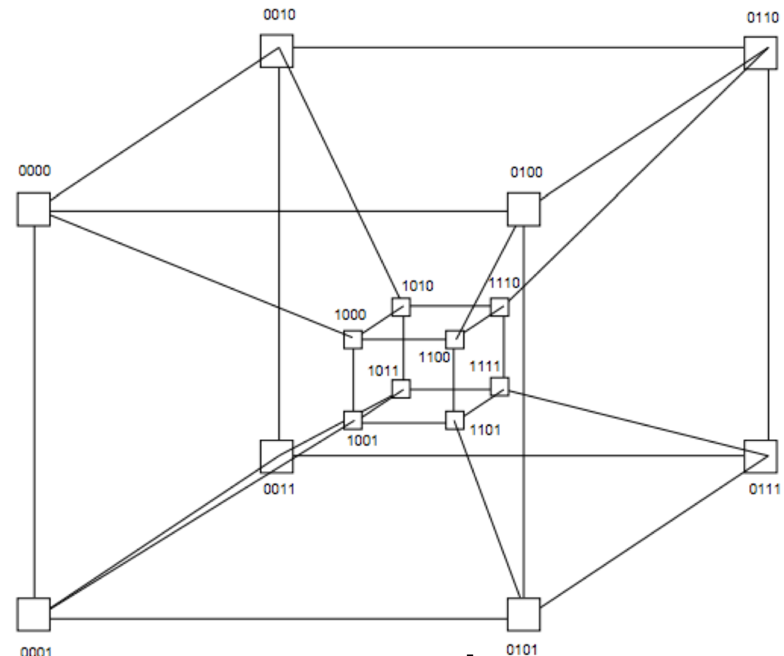
Network topology

Bus

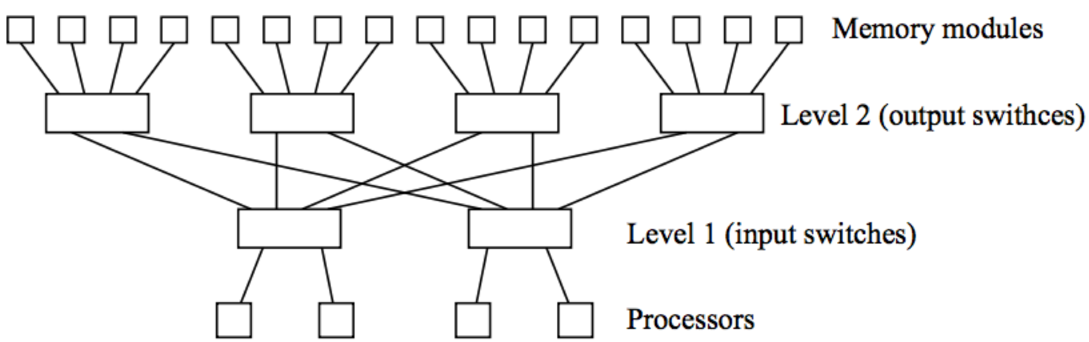


Mesh

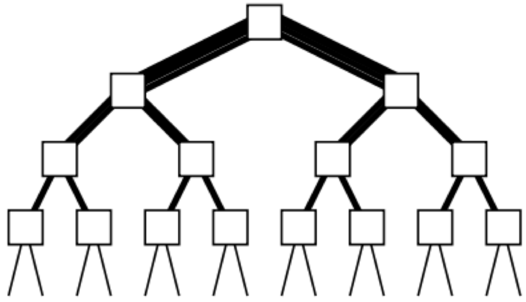




Hypercube



2-level multistage network



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

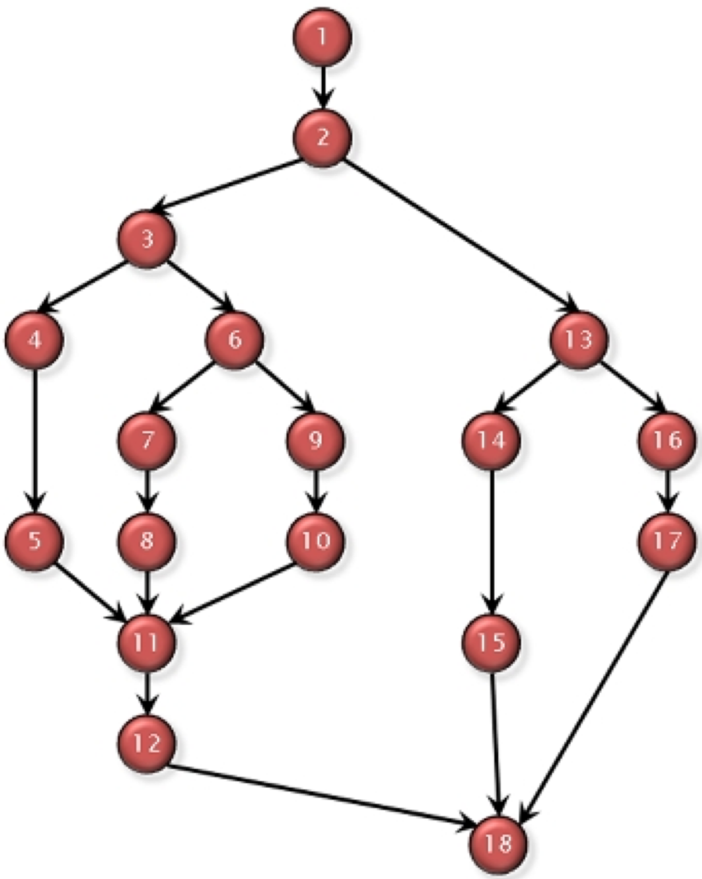
PRAM Model

- 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

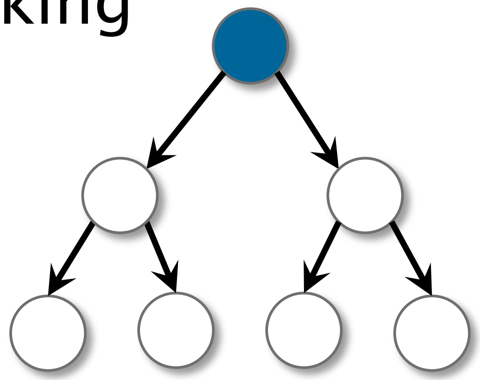


- Work = number of vertices in graph (number of operations)
- Span (Depth) = longest directed path in graph (dependence length)
- Parallelism = Work / Span
- A work-efficient parallel algorithm has work that asymptotically matches the best sequential algorithm for the problem

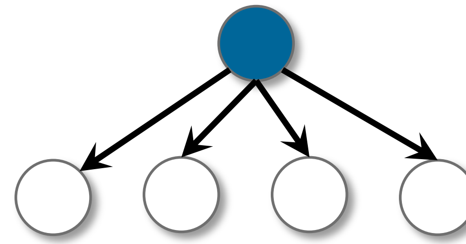
Goal: work-efficient and low (polylogarithmic) span parallel algorithms

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 6.172
  - 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

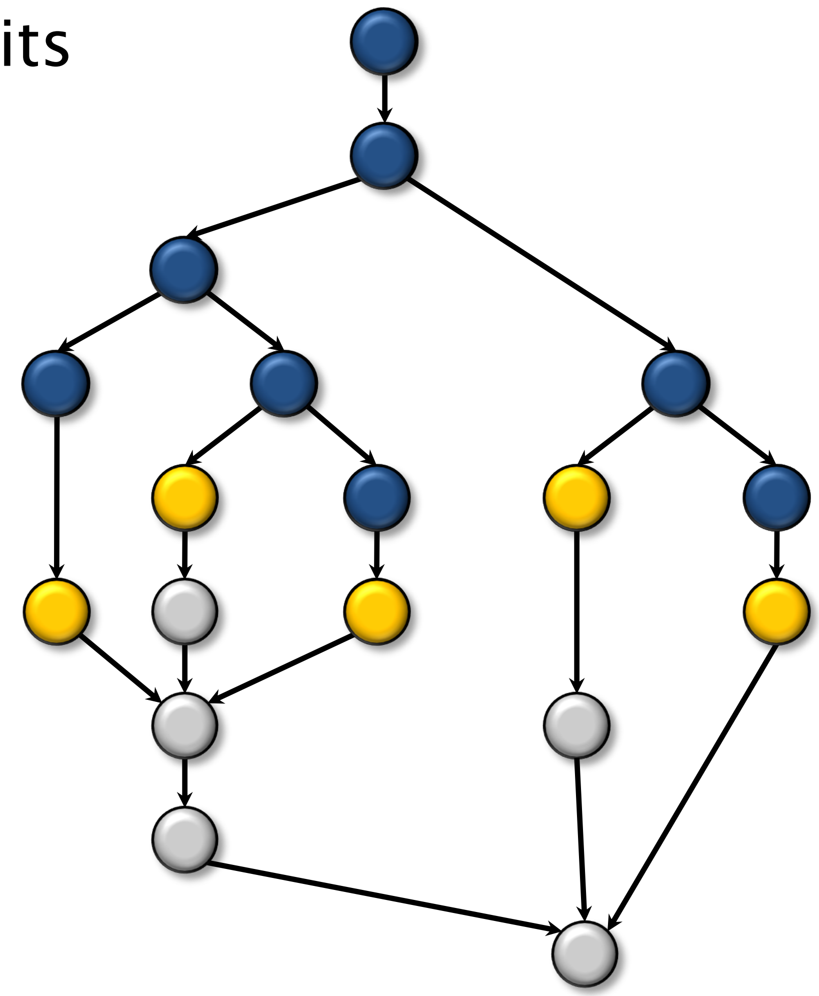
$$\text{Running time} \leq W/P + S$$

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

Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition. A task is ready if all its predecessors have executed.



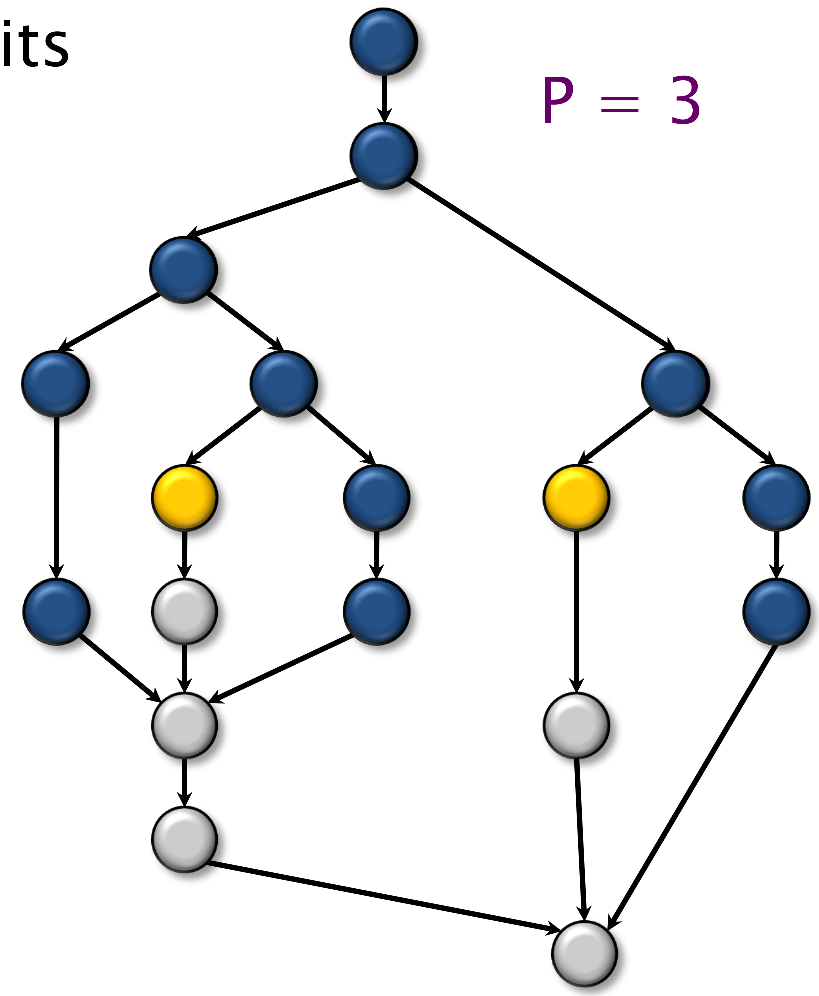
Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition. A task is ready if all its predecessors have executed.

Complete step

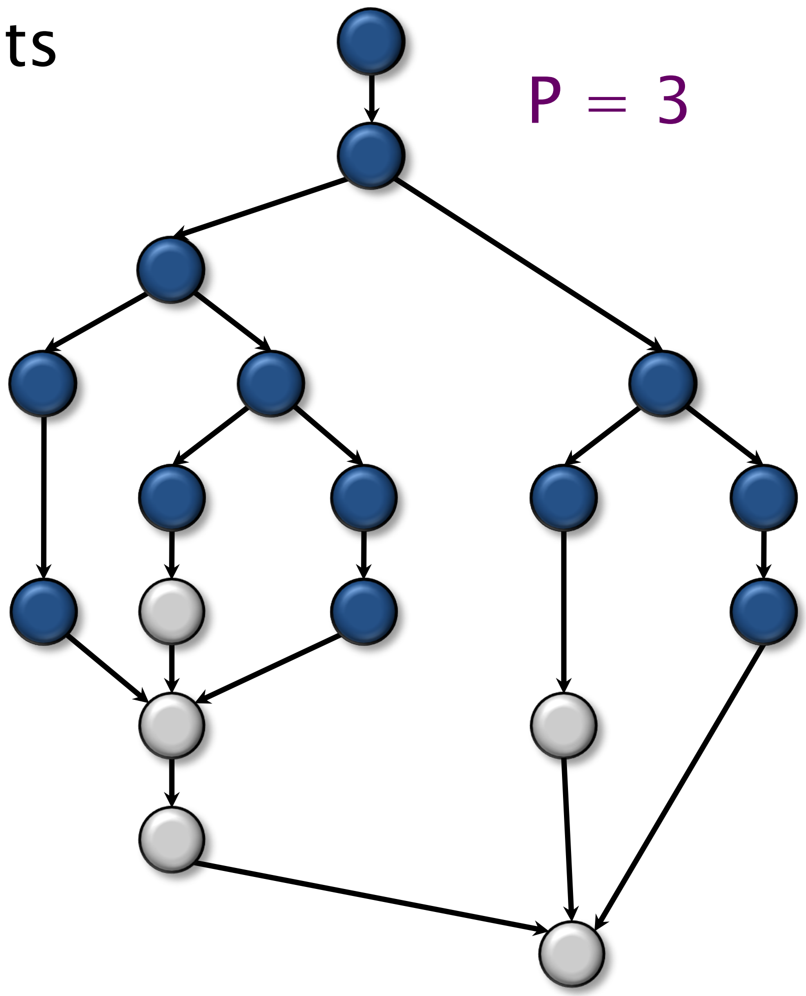
- $\geq P$ tasks ready.
- Run any P.



Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition. A task is *ready* if all its predecessors have executed.



Complete step

- $\geq$ P tasks ready.
- Run any P.

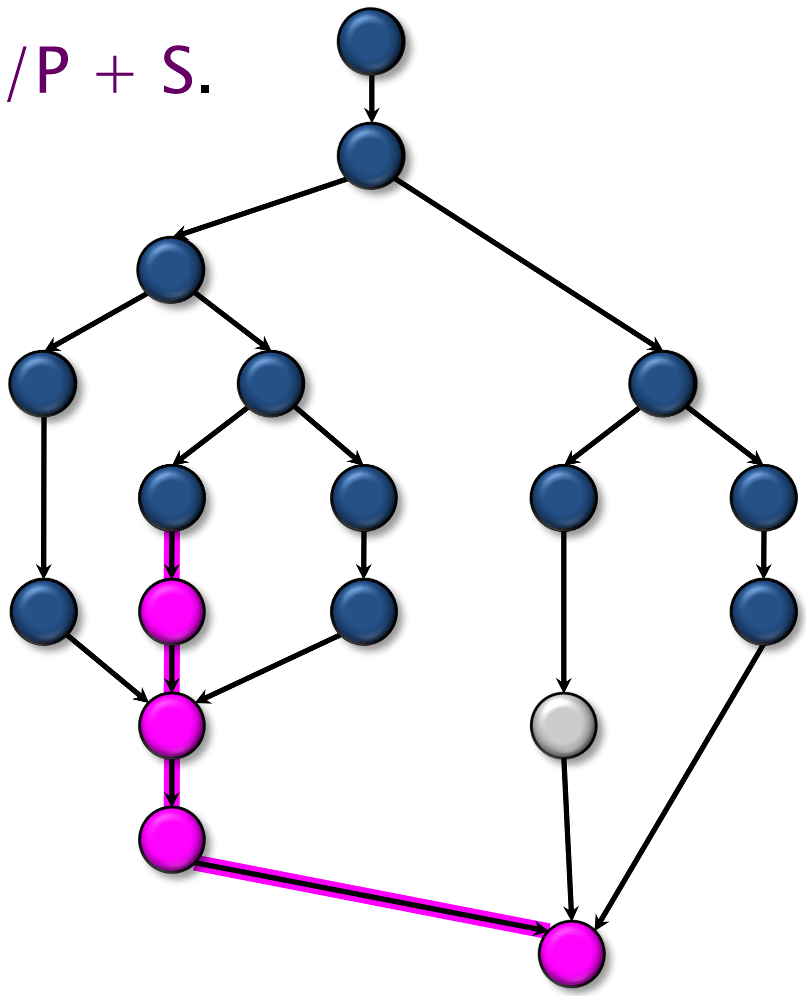
Incomplete step

- $<$ P tasks ready.
- Run all of them.

Analysis of Greedy

Theorem [G68, B75, EZL89]. Any greedy scheduler achieves

$$\text{Running Time} \le W/P + S.$$



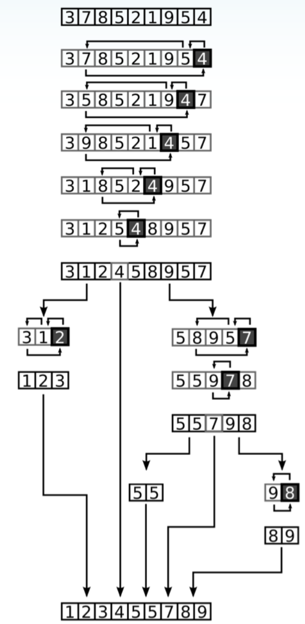
Proof.

- # complete steps $\le W/P$, since each complete step performs P work.
- # incomplete steps $\le S$, since each incomplete step reduces the span of the unexecuted dag by 1. ■

Cilk Scheduling

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

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



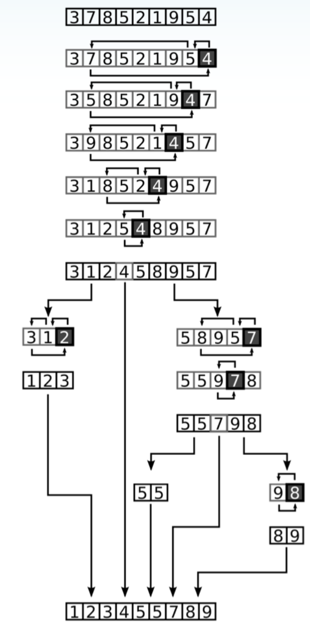
PARALLEL SUM

Parallel Sum

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

What is the span?
$S(n) = S(n/2)+O(1)$
$S(1) = O(1)$
→ $S(n) = O(\log n)$

What is the work?
$W(n) = W(n/2)+O(n)$
$W(1) = O(1)$
→ $W(n) = O(n)$


3 7 8 5 2 1 9 5 4
3 7 8 5 2 1 9 5 4
3 5 8 5 2 1 9 4 7
3 9 8 5 2 1 4 5 7
3 1 8 5 2 4 9 5 7
3 1 2 5 4 8 9 5 7
3 1 2 4 5 8 9 5 7
3 1 2
5 8 9 5 7
1 2 3
5 5 9 7 8
5 5 7 9 8
5 5
9 8
8 9
1 2 3 4 5 5 7 8 9

PREFIX SUM

Prefix Sum

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

- 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)

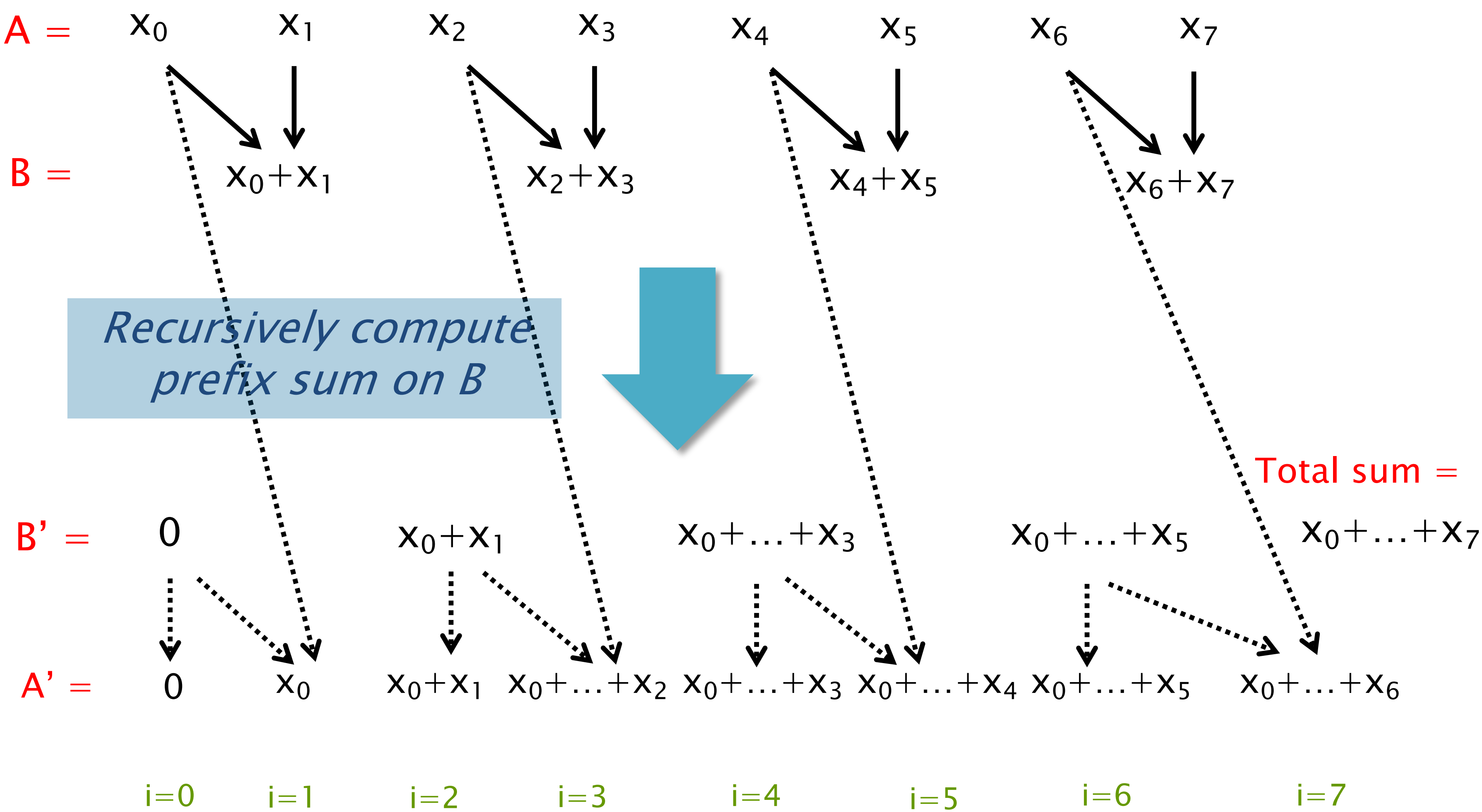
Sequential Prefix Sum

```
Input: array A of length n
Output: array A' and total sum

cumulativeSum = 0;
for i=0 to n-1:
   A'[i] = cumulativeSum;
   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



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

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

Total sum = $x_0+\ldots+x_7$

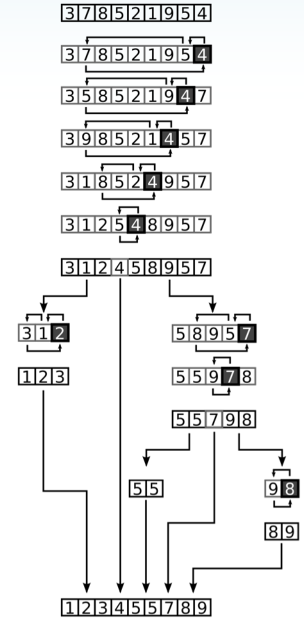
Parallel Prefix Sum

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', sum) = PrefixSum(B, n/2)
  for i=0 to n-1 in parallel:
     if (i mod 2) == 0:   A'[i] = B'[i/2]
     else:   A'[i] = B'[(i-1)/2] + A[i-1]
  return (A', sum)
```

What is the span?
$S(n) = S(n/2)+O(1)$
$S(1) = O(1)$
→ $S(n) = O(\log n)$

What is the work?
$W(n) = W(n/2)+O(n)$
$W(1) = O(1)$
→ $W(n) = O(n)$


378521954
378521954
358521947
398521457
318524957
312548957
312458957
312
58957
123
55978
55798
55
98
89
123455789

FILTER

Filter

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

A =

2	4	3	1	3

B =

T	F	T	T	F

A' =

2	3	1

- 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

Prefix sum

B' =

0	1	1	2	3

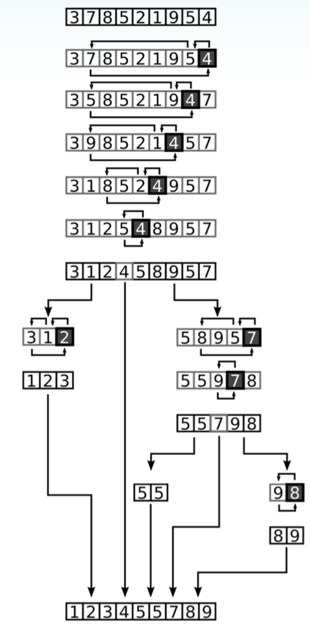
Total sum = 3

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

Allocate array of size 3

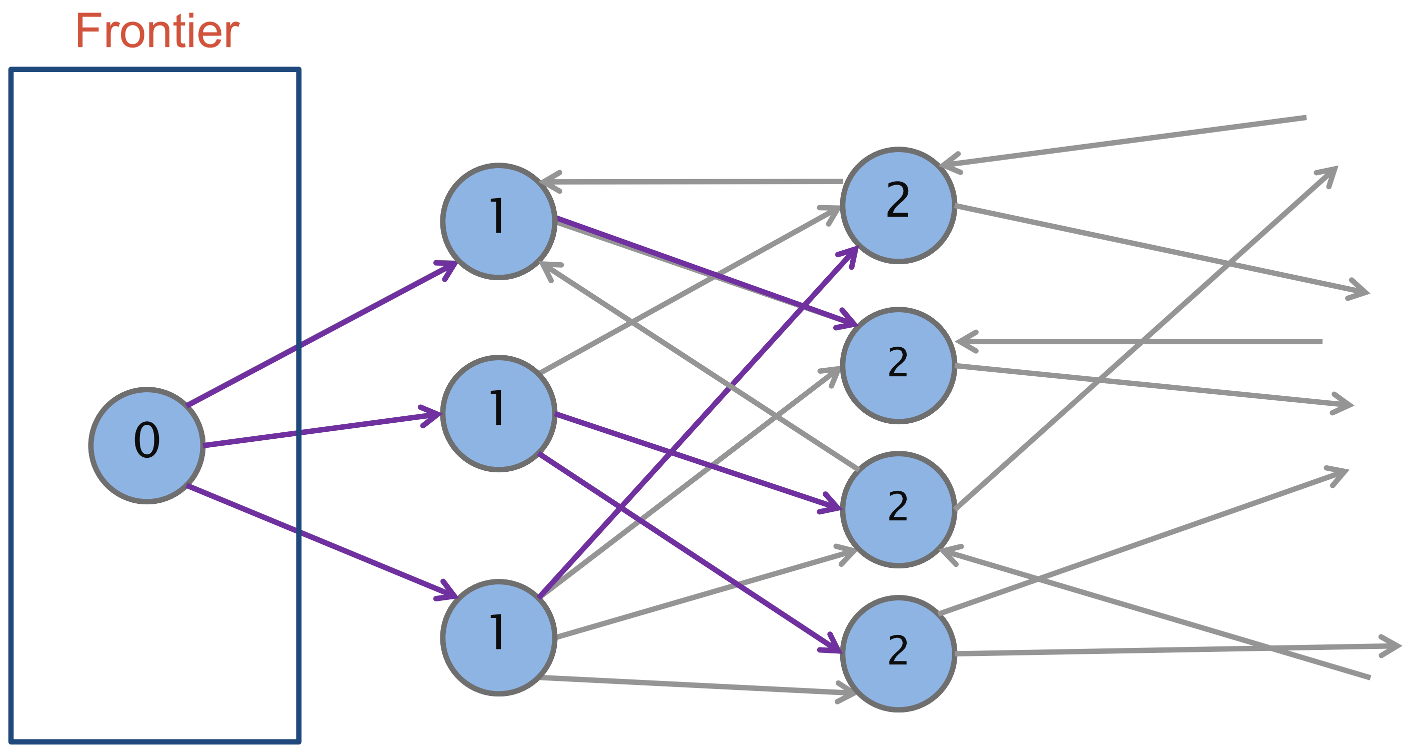
A' =

2	3	1



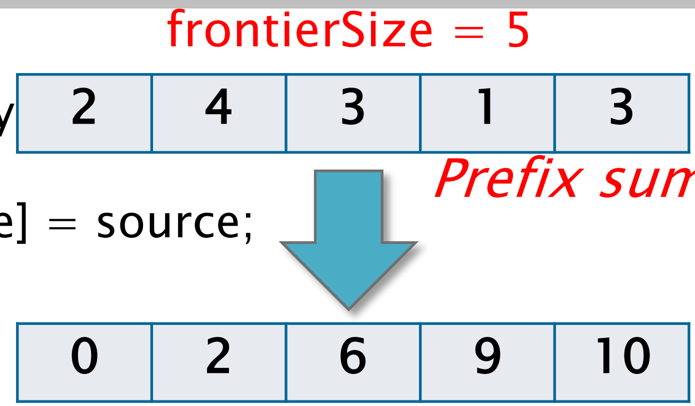
PARALLEL BREADTH-FIRST SEARCH

Parallel BFS Algorithm



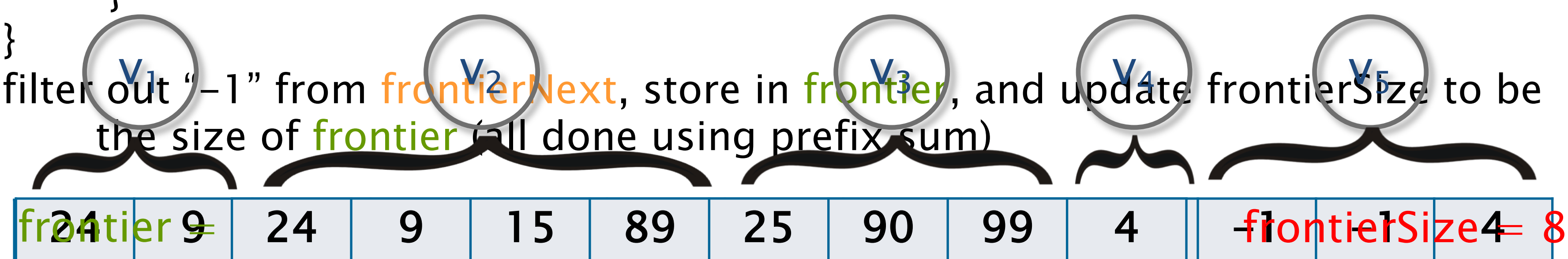
- Can process each frontier in parallel
  - Parallelize over both the vertices and their outgoing edges

Parallel BFS Code




```
BFS(Offsets, Edges, source) {
  parent, frontier, frontierNext, and degrees are arrays
  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)
  }
}
```



BFS Work-Span Analysis

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

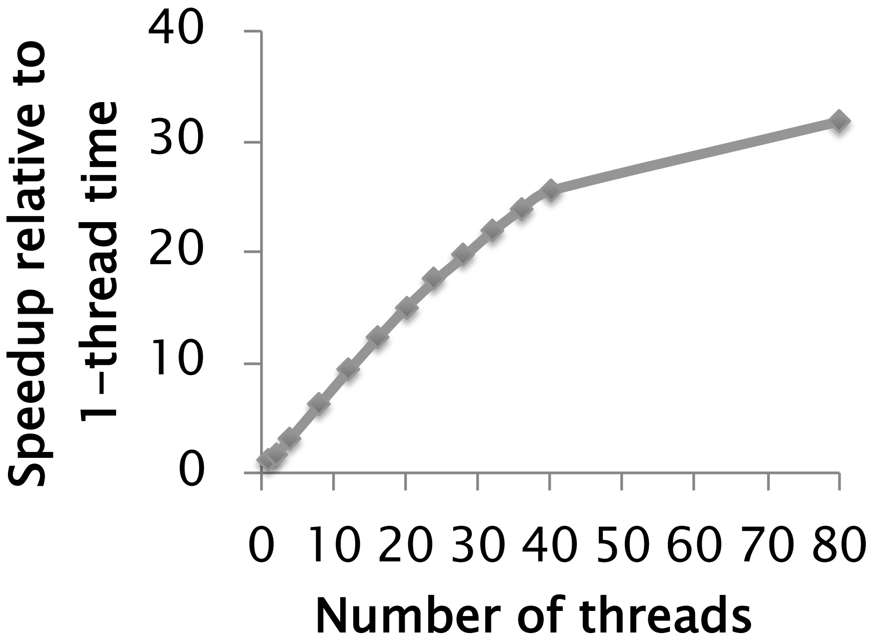
Span = O(Δ log m)

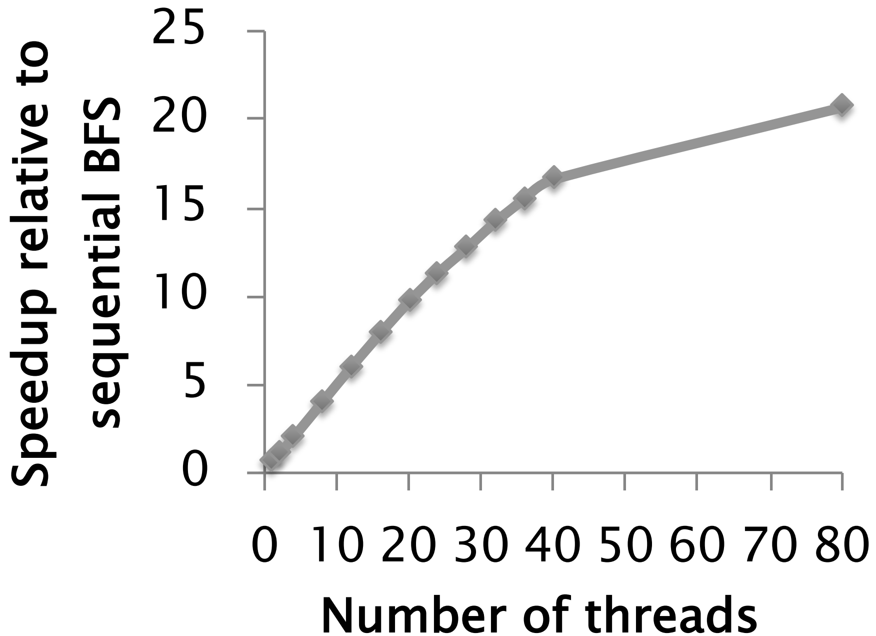
- Sum of frontier sizes = n
- Each edge traversed once -> m total visits
- Work of prefix sum on each iteration is proportional to frontier size -> Θ(n) total
- Work of filter on each iteration is proportional to number of edges traversed -> Θ(m) total

Work = Θ(n+m)

Performance of Parallel BFS

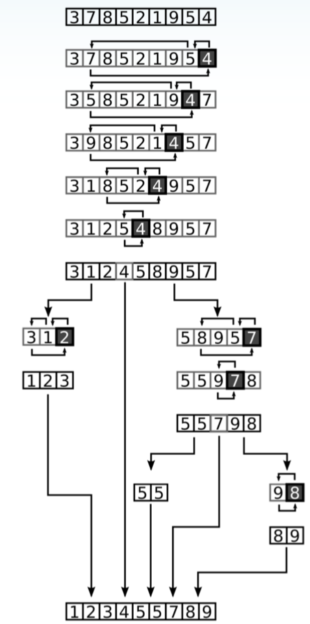
- Random graph with $n=10^7$ and $m=10^8$
  - 10 edges per vertex
- 40-core machine with 2-way hyperthreading





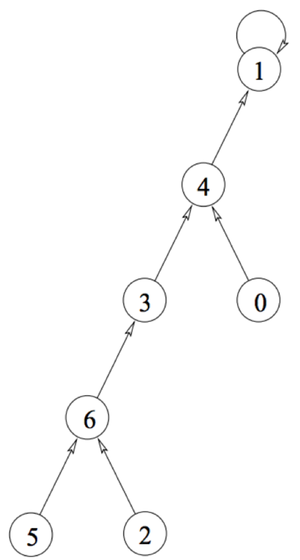
- 31.8x speedup on 40 cores with hyperthreading
- Sequential BFS is 54% faster than parallel BFS on 1 thread

POINTER JUMPING AND LIST RANKING

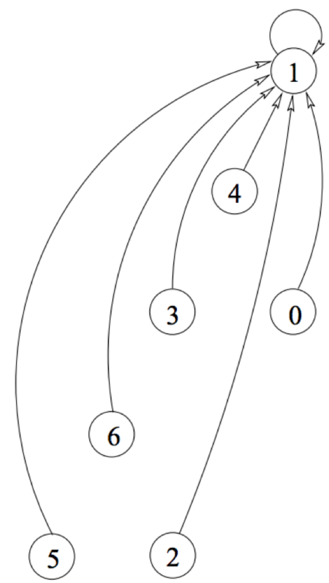


Pointer Jumping

- 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 $P = [1, 1, 1, 1, 1, 1, 1]$. ;eration

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

What is the work and span?

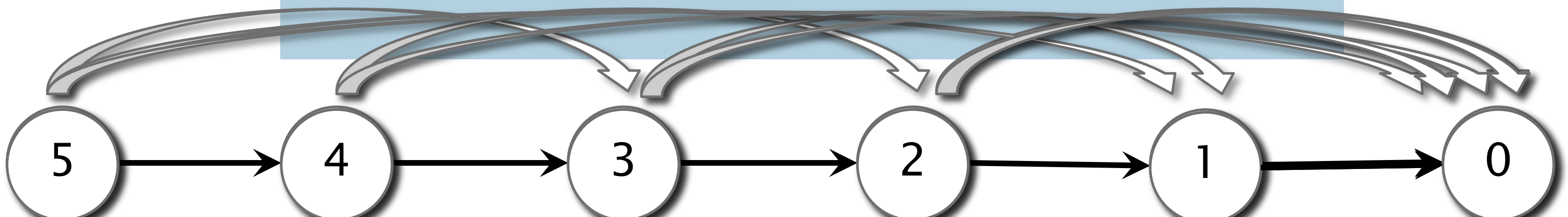
W = O(n log n)
S = O(log n)

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 = rank[P[i]];
        temp2 = P[P[i]];
    parallel-for i=0 to n-1:
        rank[i] = rank[i] + temp;
        P[i] = temp2;
```



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?

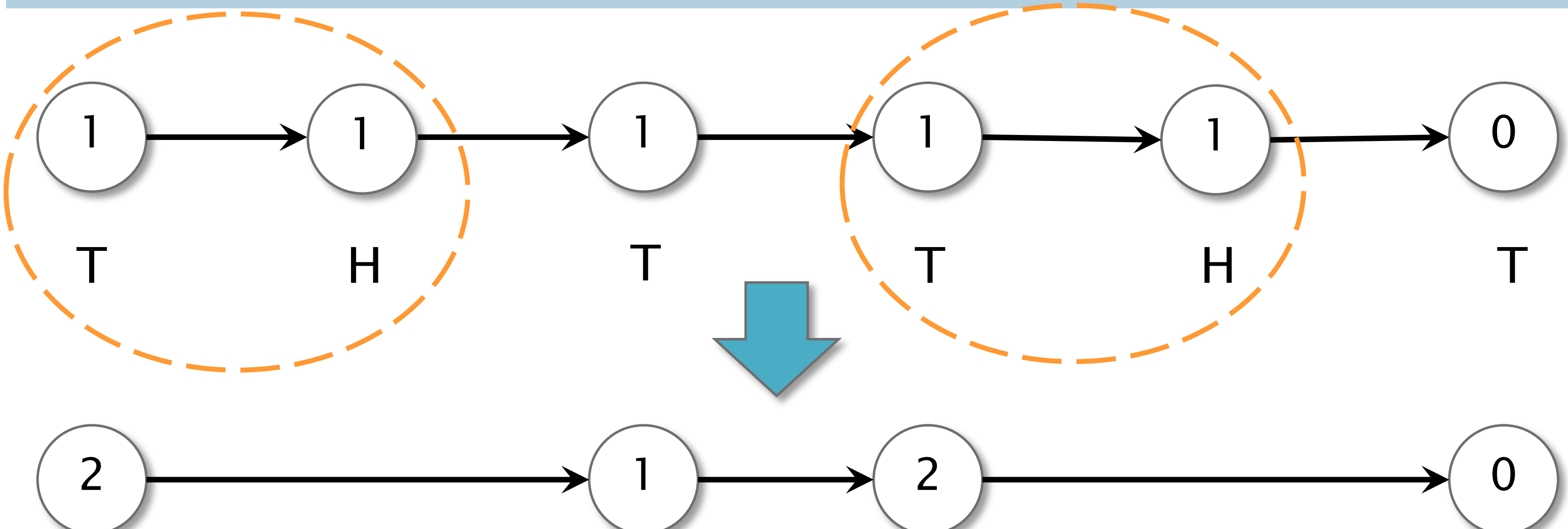
$W = O(n \log n)$
$S = O(\log n)$

Sequential algorithm only requires O(n) work

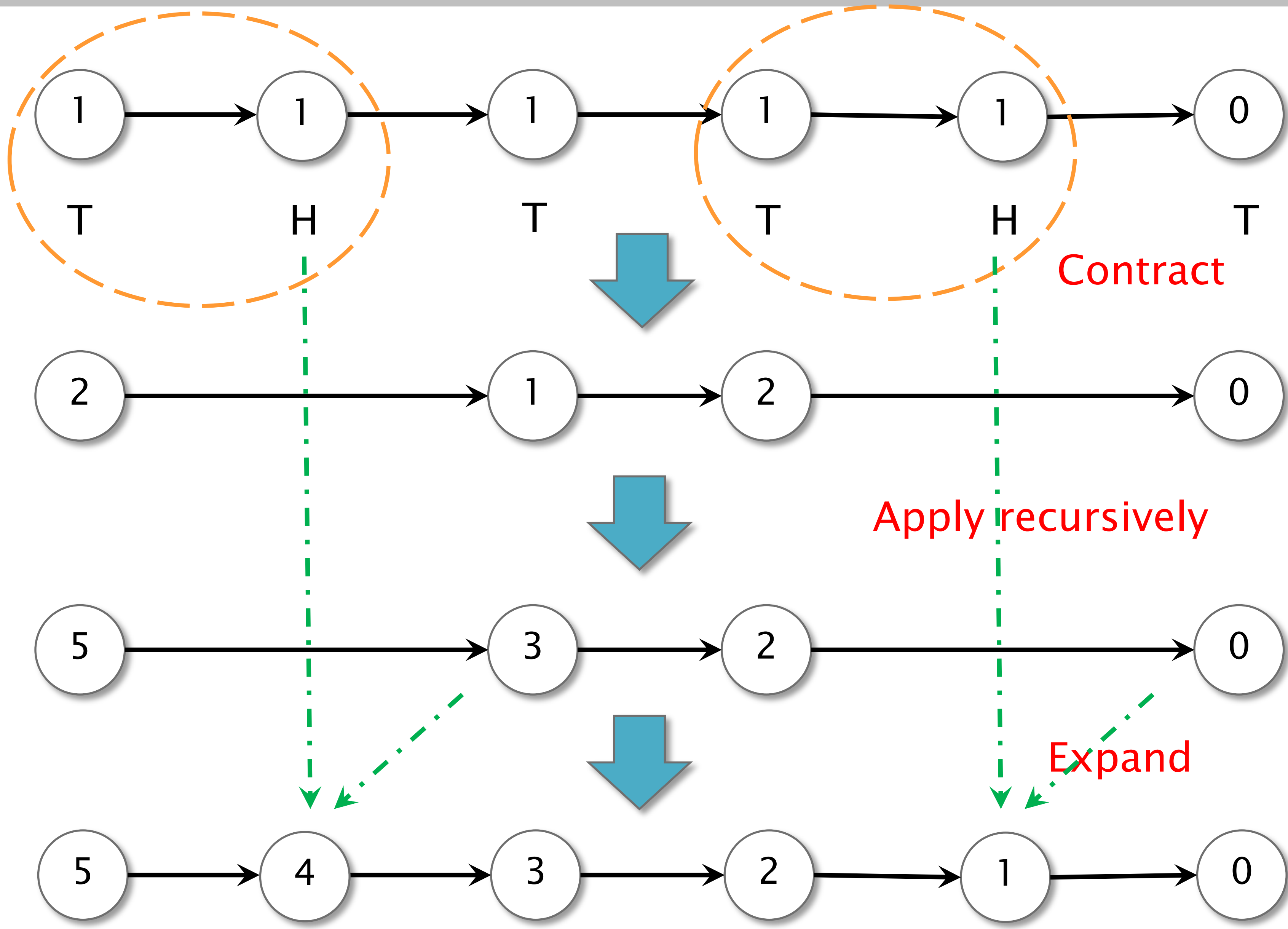
Work-Efficient List Ranking

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. rank[u] = rank[u]+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 rank[v] = rank[v] + rank[P[v]]



Work-Efficient List Ranking

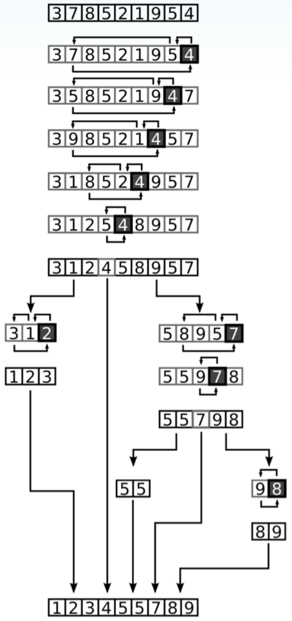


Work-Span Analysis

- 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



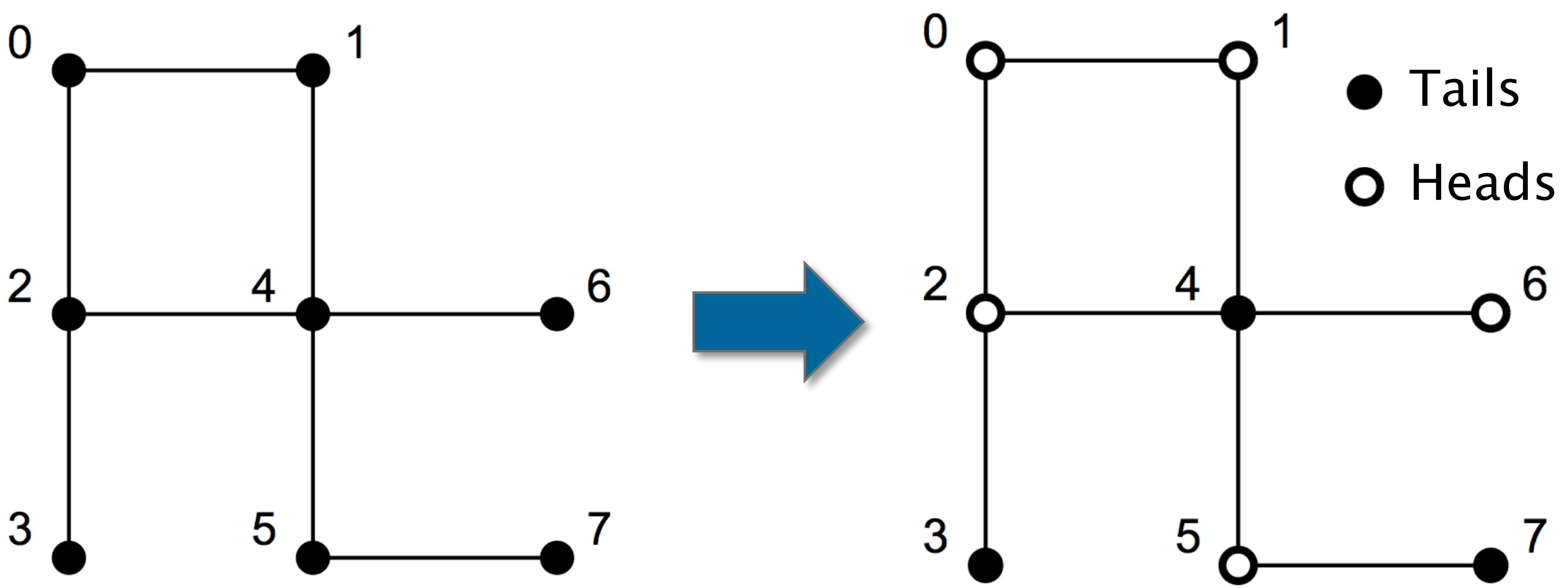
CONNECTED COMPONENTS

Connected Components

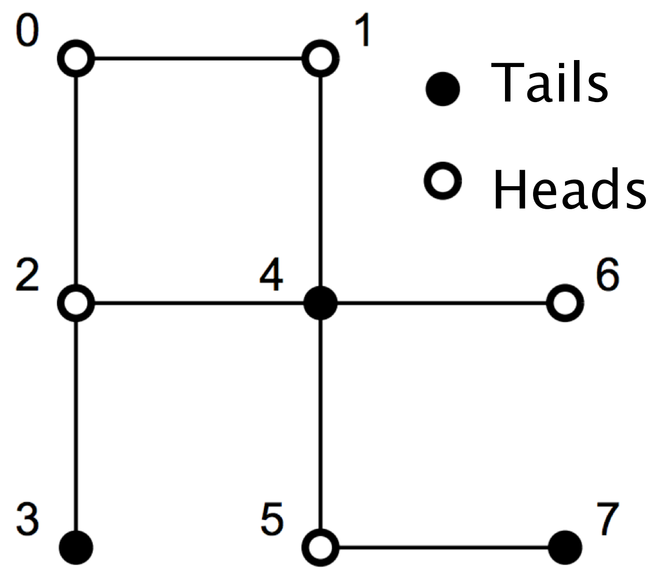
- Given an undirected graph, label all vertices such that L(u) = L(v) if and only if there is a path between u and v
- BFS span is proportional to diameter
  - Works well for graphs with small diameter
- Today we will see a randomized algorithm that takes O((n+m)log n) work and O(log n) span
  - Deterministic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

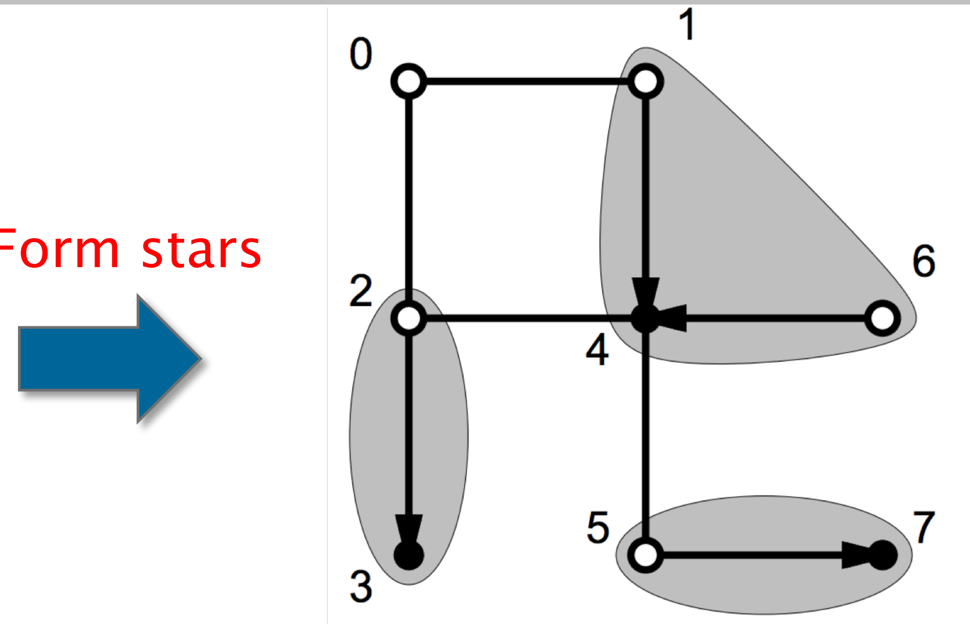
Random Mate

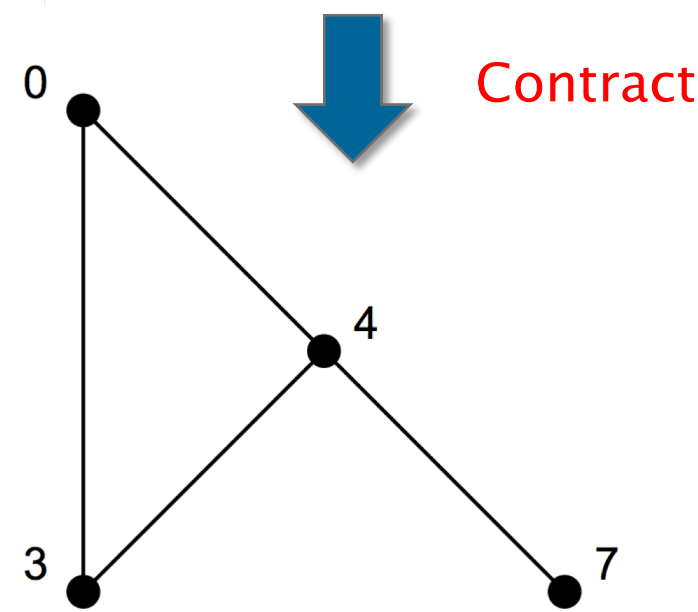
- Idea: Form a set of non-overlapping star subgraphs and contract them
- Each vertex flips a coin. For each Heads vertex, pick an arbitrary Tails neighbor (if there is one) and point to it



Random Mate







Repeat until each component has a single vertex

Expand vertices back in reverse order with label of neighbor

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 coin(v)=Heads, hook to arbitrary Tails neighbor
       w and set L(v) = w
    3. E’ = { (L(u),L(v)) | (u,v) ∈ E and L(u) ≠ L(v) }
    4. L’ = CC_Random_Mate(L, E’)
    5. For v where coin(v)=Heads, set L’(v) = L’(w) where w is the
       Tails neighbor that v hooked to in Step 2
    6. Return L’
```

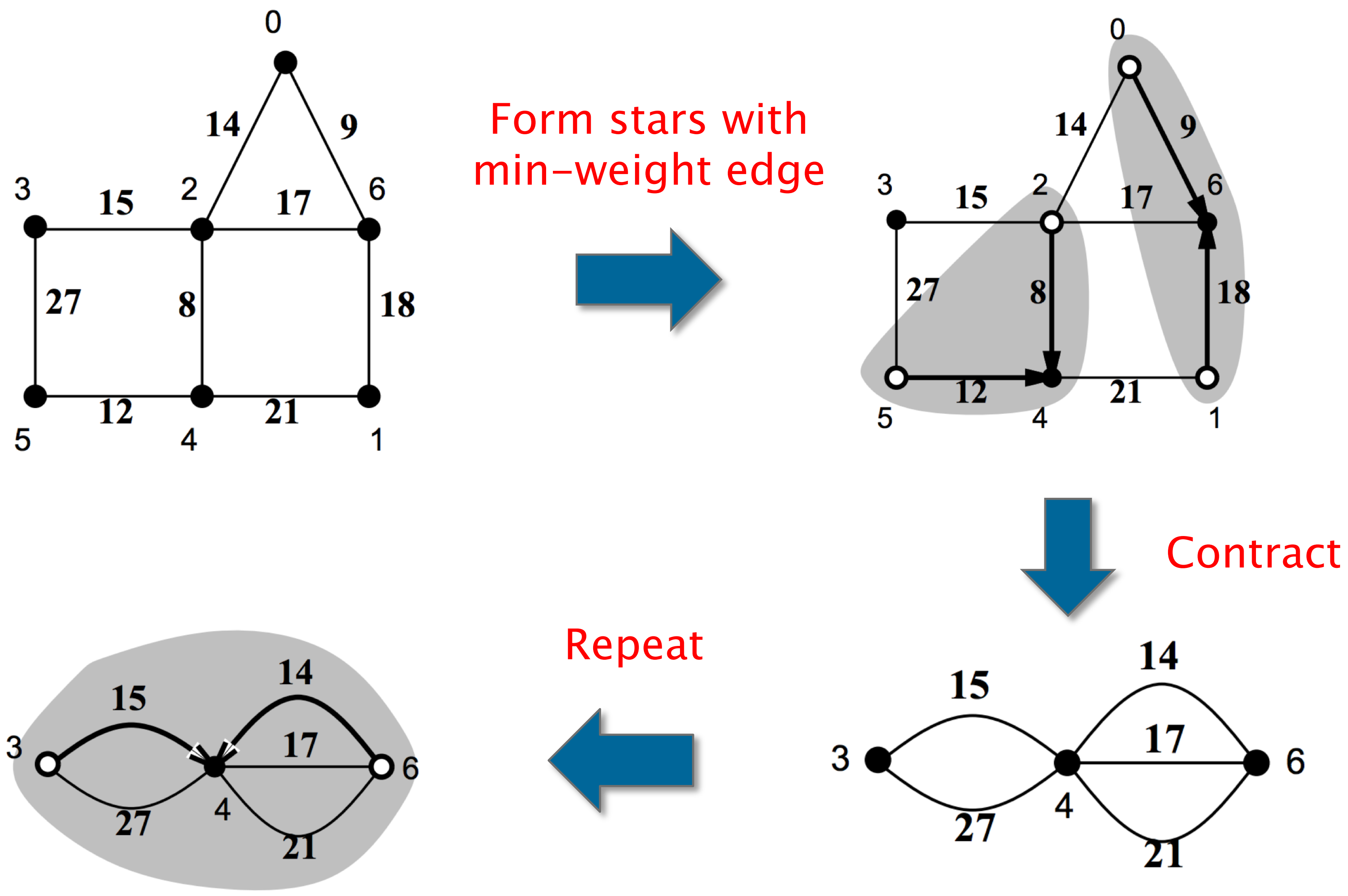
- Each iteration requires O(m+n) work and O(1) span
  - Assumes we do not pack vertices and edges
- Each iteration eliminates 1/4 of the vertices in expectation

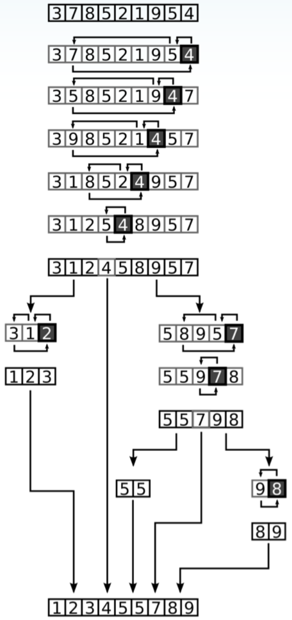
W = O((m+n)log n) w.h.p. S = O(log n) w.h.p.

(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





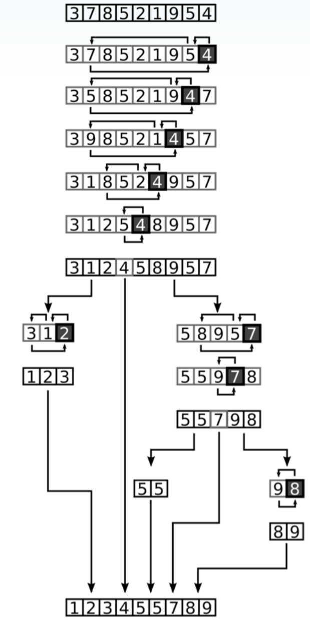
PARALLEL BELLMAN-FORD

Bellman-Ford Algorithm

```
Bellman-Ford(G, source):
    ShortestPaths = {∞, ∞, …, ∞}    //size n; stores shortest path distances
    ShortestPaths[source] = 0
    for i=1 to n-1:
parallel for each vertex v in G:
        parallel for each w in neighbors(v):
                                                        concurrent write
                writeMin(&ShortestPaths[w], ShortestPaths[v] + weight(v,w))
        if no shortest paths changed:
            return ShortestPaths
    report "negative cycle"
```

- What is the work and span assuming writeMin has unit cost?
- Work = O(mn)
- Span = O(n)

QUICKSORT



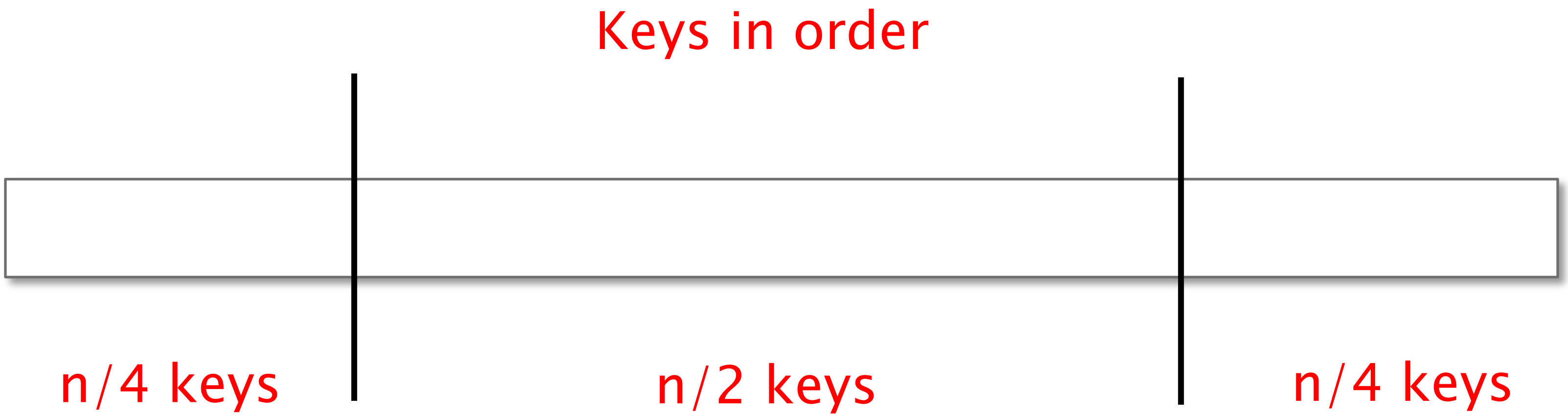
378521954
378521954
358521947
398521457
318524957
312548957
312458957
312
58957
123
55978
55798
55
98
89
123455789

Parallel Quicksort

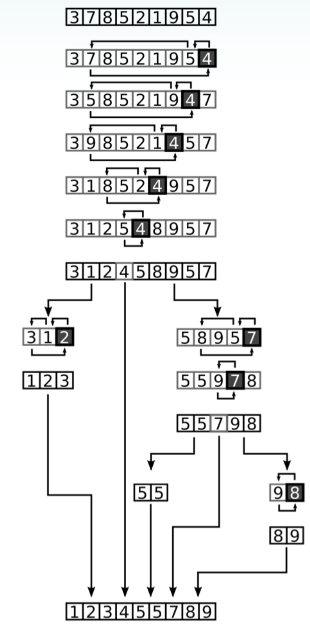
```
static void quicksort(int64_t *left, int64_t *right)
{
  int64_t *p;
  if (left == right) return;
  p = partition(left, right);
  cilk_spawn quicksort(left, p);
  quicksort(p + 1, right);
  cilk_sync;
}
```

- Partition picks random pivot p and splits elements into left and right subarrays
- Partition can be implemented using prefix sum in linear work and logarithmic span
- Overall work is O(n log n)
- What is the span?

Parallel Quicksort Span



- Pivot is chosen uniformly at random
- 1/2 chance that pivot falls in middle range, in which case sub-problem size is at most 3n/4
- Expected span:
  - $S(n) \leq (1/2)\, S(3n/4) + O(\log n)$
    $= O(\log^2 n)$
- Can get high probability bound with Chernoff bound


378521954
378521954
358521947
398521457
318524957
312548957
312458957
312
58957
123
55978
55798
55
98
89
123455789

RADIX SORT

Radix Sort

- Consider 1–bit digits

```
Radix_sort(A, b) //b is the number of bits of A
  For i from 0 to b-1: //sort by i'th most significant bit
        Flags = { (a >> i) mod 2 | a ∈ A }
        NotFlags = { !(a >> i) mod 2 | a ∈ A}
        (sum0, R0) = prefixSum(NotFlags)
        (sum1, R1) = prefixSum(Flags)
        Parallel-for j = 0 to |A|-1:
              if(Flags[j] = 0):  A'[R0[j]] = A[j]
              else:       A'[R1[j]+sum0] = A[j]
        A = A'
```

A =	1	2	6	5	4	3
Flags =	1	0	0	1	0	1
NotFlags =	0	1	1	0	1	0
A' =	2	6	4	1	5	3

R_1 =	0	1	1	1	2	2
R_0 =	0	0	1	2	2	3

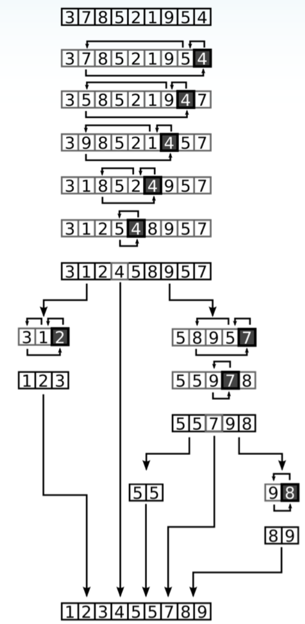
$sum_0 = 3$

- Each iteration is stable

Work-Span Analysis

```
Radix_sort(A, b) //b is the number of bits of A
  For i from 0 to b-1:
        Flags = { (a >> i) mod 2 | a ∈ A }
        NotFlags = { !(a >> i) mod 2 | a ∈ A}
        (sum_0, R_0) = prefixSum(NotFlags)
        (sum_1, R_1) = prefixSum(Flags)
        Parallel-for j = 0 to |A|-1:
                if(Flags[j] = 0):  A'[R_0[j]] = A[j]
                else:      A'[R_1[j]+sum_0] = A[j]
        A = A'
```

- Each iteration requires O(n) work and O(log n) span
- Overall work = O(bn)
- Overall span = O(b log n)



REMOVING DUPLICATES

Removing Duplicates with Hashing

- Given an array A of n elements, output the elements in A excluding duplicates

Construct a table T of size m, where m is the next prime after 2n
i = 0
While (|A| > 0)

1. Parallel-for each element j in A try to insert j into T at location (hash(A[j],i) mod m) *//if the location was empty at the beginning of round i, and there are concurrent writes then an arbitrary one succeeds*
2. Filter out elements j in A such that T[(hash(A[j],i) mod m)] = A[j]
3. i = i+1

- Use a new hash function on each round
- Claim: Every round, the number of elements decreases by a factor of 2 in expectation

W = O(n) expected　　　S = $O(\log^2 n)$ w.h.p.