Algorithms at Scale

(Week 10)

Summary

Today: Parallelism

Models of Parallelism

- How to predict the performance of algorithms?

Some simple examples…

Sorting

- Parallel MergeSort

Trees and Graphs

Last Week: Caching

Breadth-First-Search

- *Sorting your graph*

MIS

- *Luby's Algorithm*
- *Cache-efficient implementation*

MST

- *Connectivity*
- *Minimum Spanning Tree*

Announcements / Reminders

Today:

MiniProject update due today.

Next week:

MiniProject explanatory section due

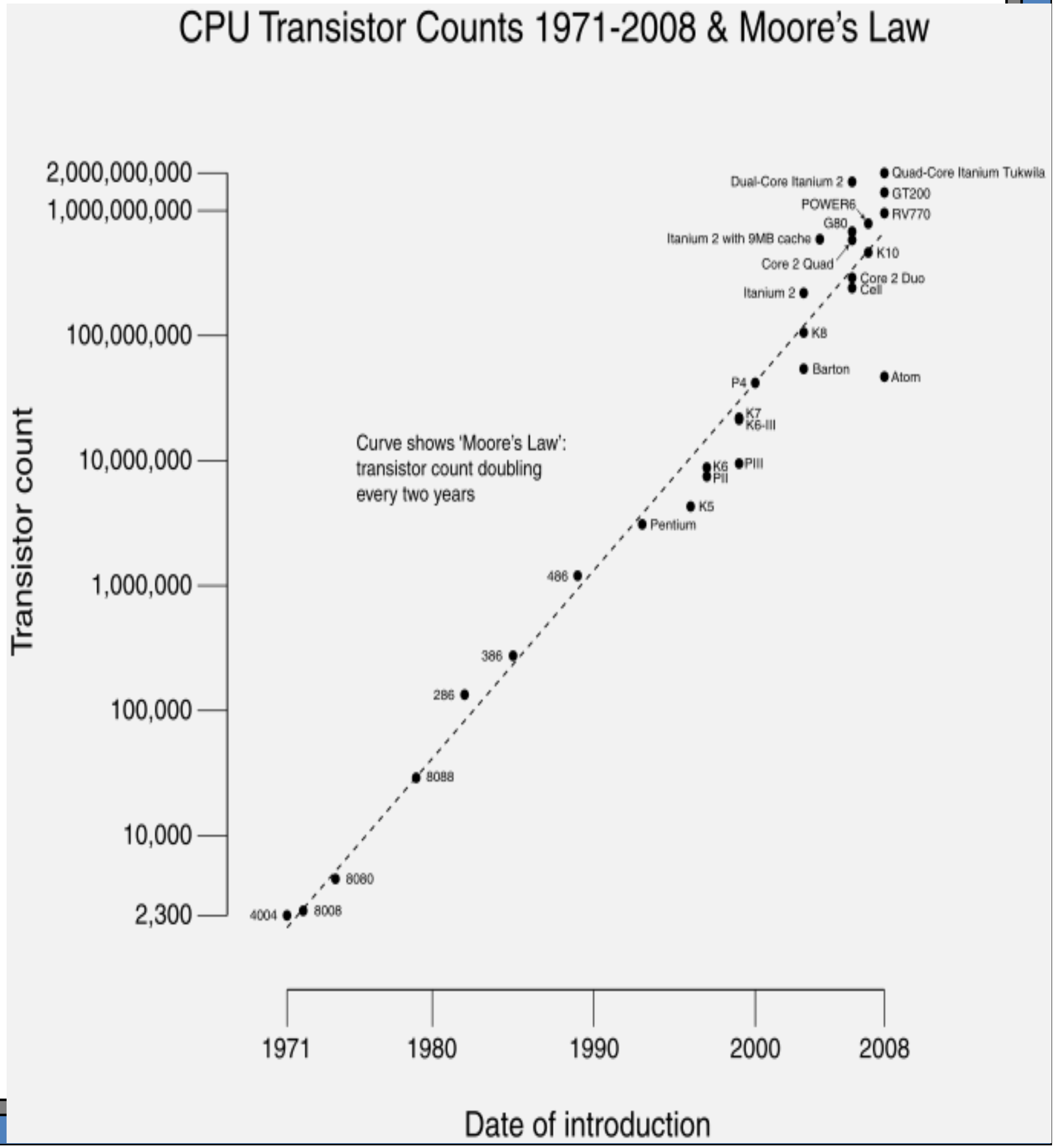
Parallel Algorithms

Moore's Law

Number of transistors doubles every 2 years!

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase." Gordon Moore, 1965

Limits will be reached in 10-20 years…maybe.



Source: Wikipedia

Parallel Algorithms

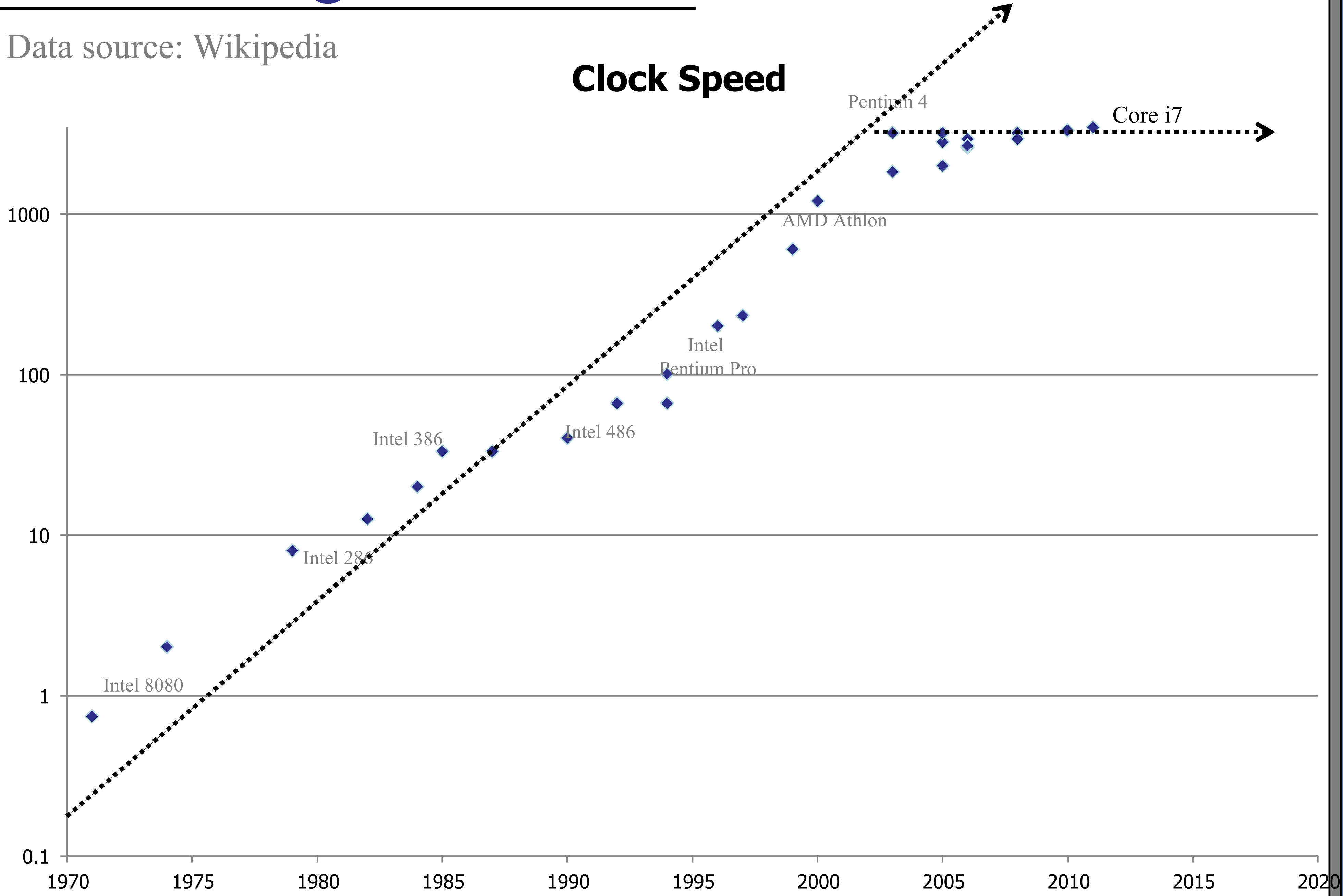
More transisters == faster computers?

- More transistors per chip → smaller transistors.
- Smaller transistors → faster
- Conclusion:

  Clock speed doubles every two years, also.

Parallel Algorithms

Data source: Wikipedia



Parallel Algorithms

What to do with more transistors?

- More functionality
  - GPUs, FPUs, specialized crypto hardware, etc.
- Deeper pipelines
- More clever instruction issue (out-of-order issue, scoreboarding, etc.)
- More on chip memory (cache)

Limits for making faster processors?

Parallel Algorithms

Problems with faster clock speeds:

- Heat
  - Faster switching creates more heat.
- Wires
  - Adding more components takes more wires to connect.
  - Wires don't scale well!
- Clock synchronization
  - How do you keep the entire chip synchronized?
  - If the clock is too fast, then the time it takes to propagate a clock signal from one edge to the other matters!

Parallel Algorithms

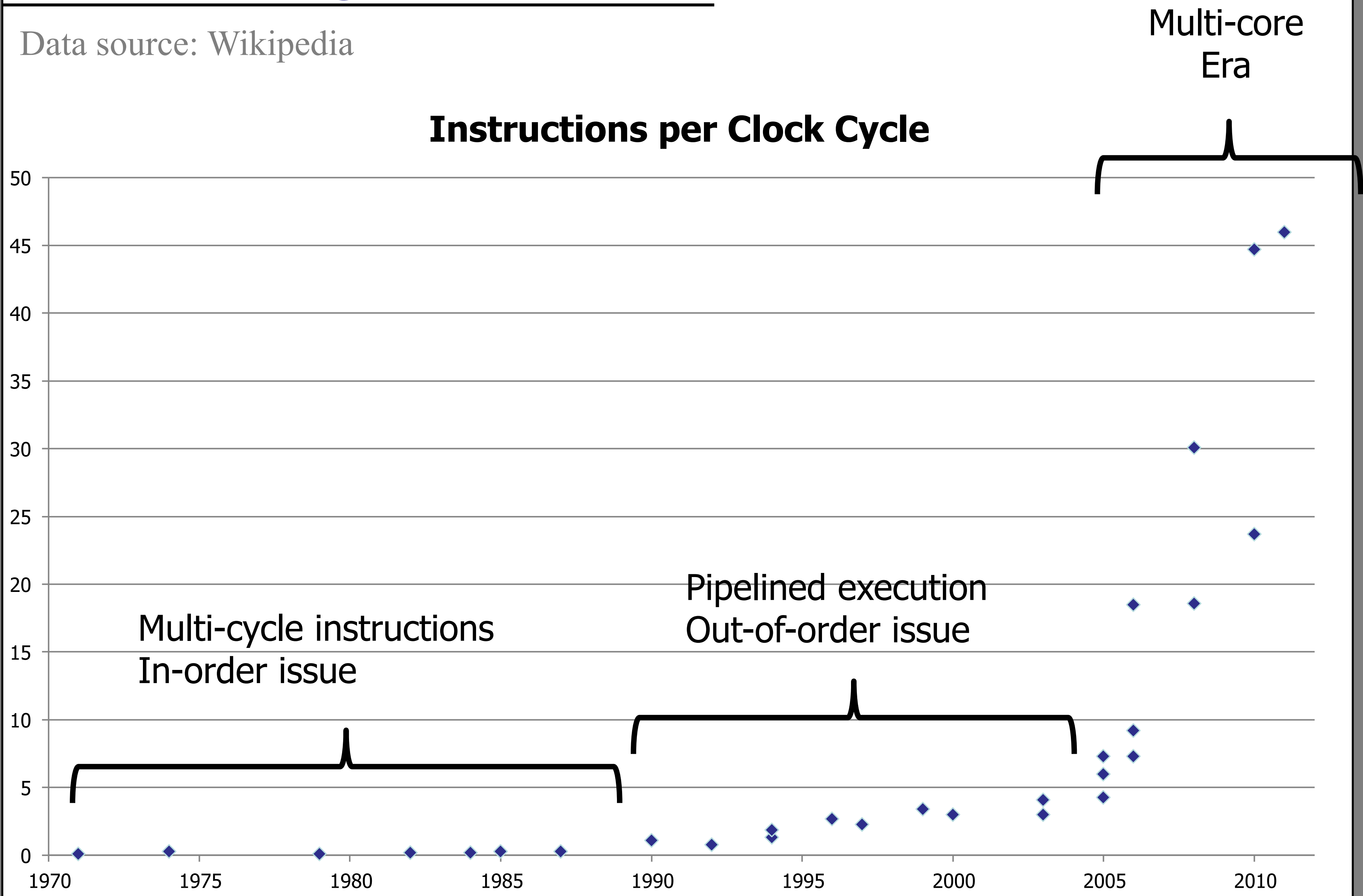
Conclusion:

- We have lots of new transistors to use.
- We can’t use them to make the CPU faster.

What do we do?

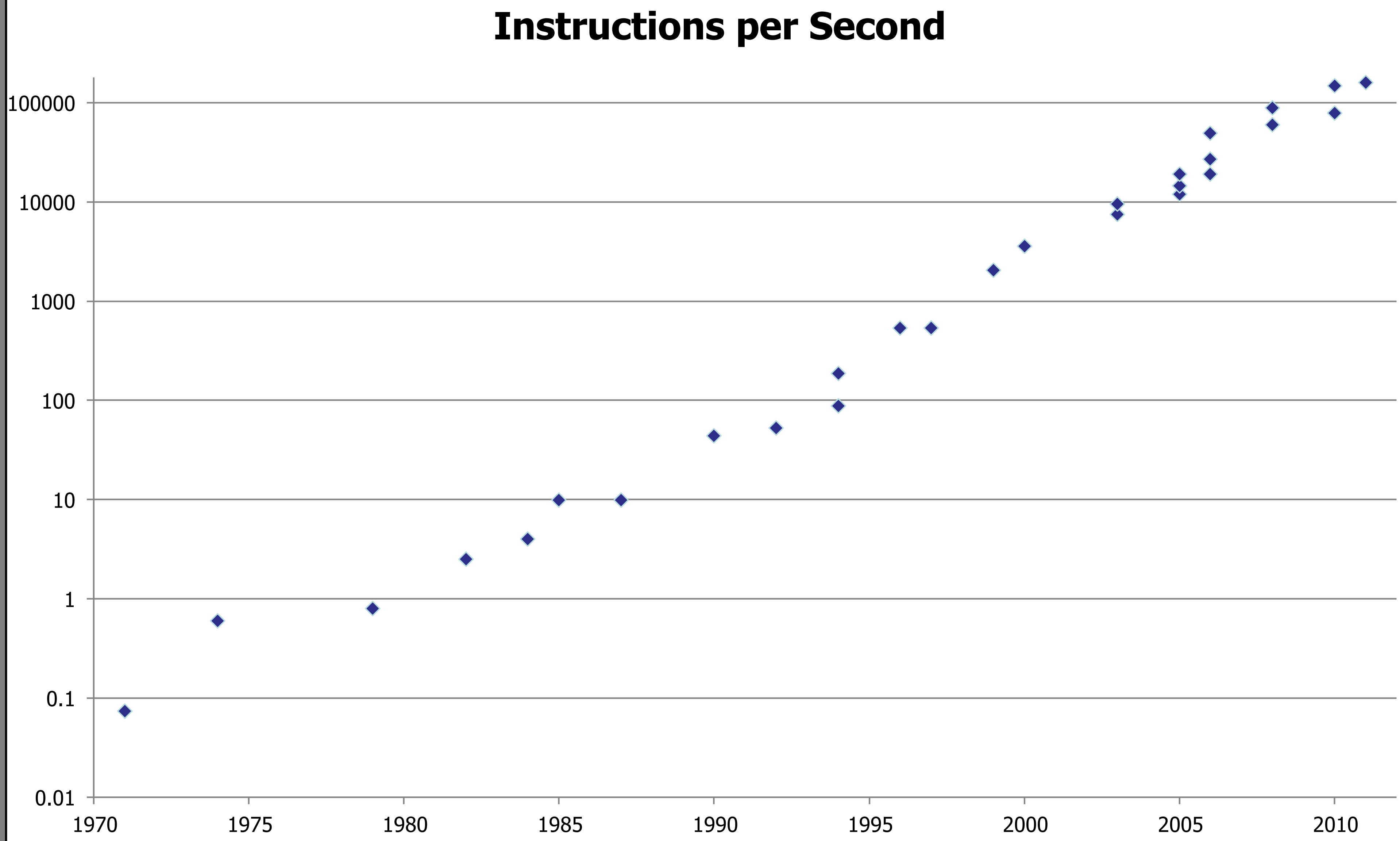
Parallel Algorithms

Data source: Wikipedia



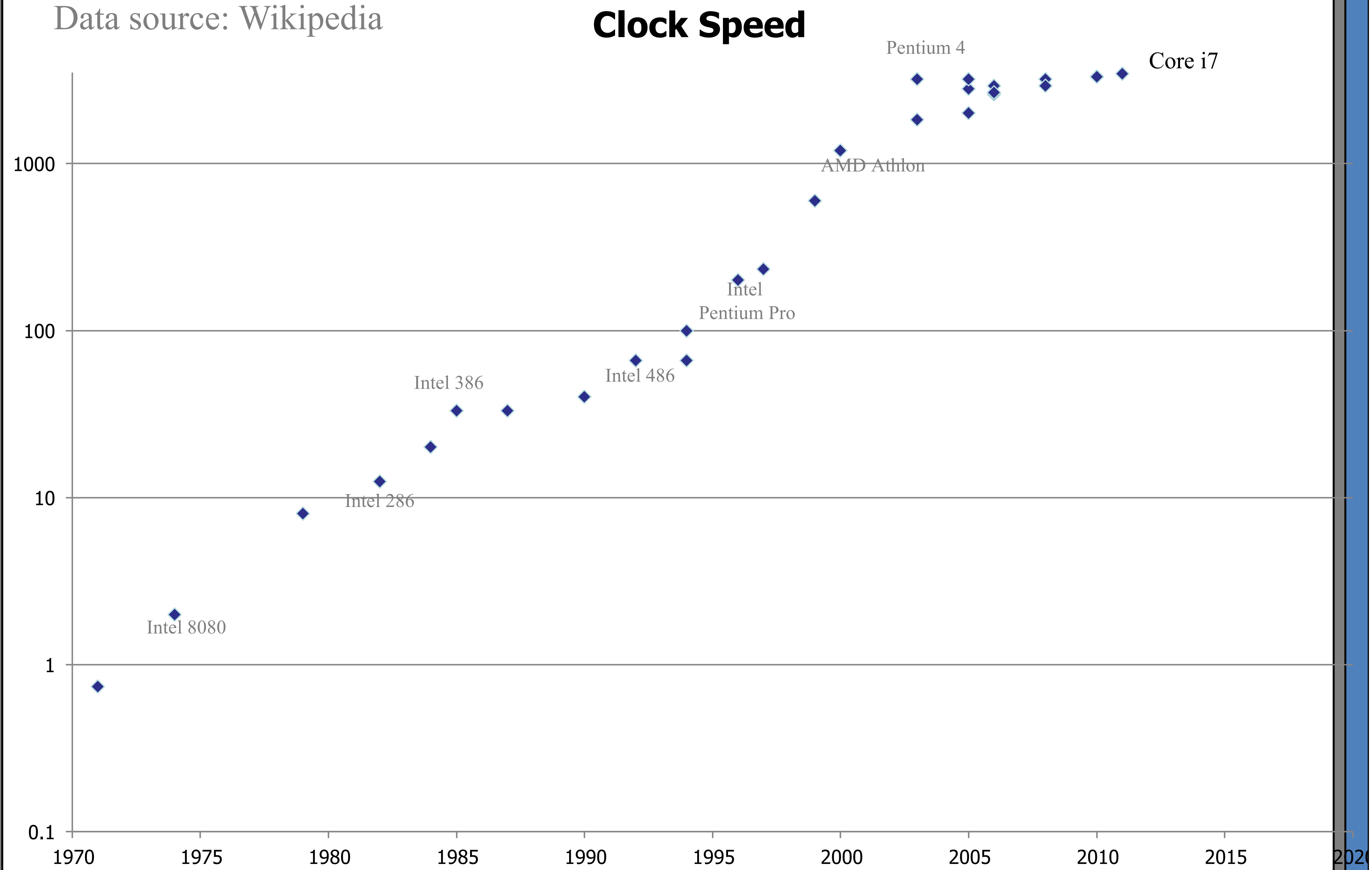
Parallel Algorithms

Data source: Wikipedia



Parallel Algorithms

Data source: Wikipedia



Parallel Algorithms

To make an algorithm run faster:

- Must take advantage of multiple cores.
- Many steps executed at the same time!

Parallel Algorithms

To make an algorithm run faster:

- Must take advantage of multiple cores.
- Many steps executed at the same time!

CS5234 algorithms:

- Sampling → lots of parallelism
- Sketches → lots of parallelism
- Streaming → lots of parallelism
- Cache-efficient algorithms??

Parallel Algorithms

Challenges:

– How do we write parallel programs?

- Partition problem over multiple cores.
- Specify what can happen at the same time.
- Avoid unnecessary sequential dependencies.
- Synchronize different threads (e.g., locks).
- Avoid race conditions!
- Avoid deadlocks!

Parallel Algorithms

Challenges:

- How do we analyze parallel algorithms?
  - Total running time depends on # of cores.
  - Cost is harder to calculate.
  - Measure of scalability?

Parallel Algorithms

Challenges:

- How do we debug parallel algorithms?
  - More non-determinacy
  - Scheduling leads to un-reproduceable bugs
    - Heisenbugs!
  - Stepping through parallel programs is hard.
  - Race conditions are hard.
  - Deadlocks are hard.

Parallel Algorithms

Different types of parallelism:

- multicore
  - on-chip parallelism: synchronized, shared caches, etc.
- multisocket
  - closely coupled, highly synchronized, shared caches
- cluster / data center
  - connected by a high-performance interconnect
- distributed networks
  - slower interconnect, less tightly synchronized

Parallel Algorithms

Different settings ➔

1) Different costs

2) Different solutions

Different types of paral

- multicore
  - on-chip parallelism:
- multisocket
  - closely coupled, highly synchronized, shared caches
- cluster / data center
  - connected by a high-performance interconnect
- distributed networks
  - slower interconnect, less tightly synchronized

Parallel Algorithms

Different types of parallelism:

Today

- multicore
  - on-chip parallelism: synchronized, shared caches, etc.
- multisocket
  - closely coupled, highly synchronized, shared caches
- cluster / data center
  - connected by a high-performance interconnect
- distributed networks
  - slower interconnect, less tightly synchronized

Parallel Algorithms

Different types of parallelism:

Today

- multicore
  - on-chip parallelism: synchronized, shared caches, etc.
- multisocket
  - closely coupled, highly synchronized, shared caches
- cluster / data center
  - connected by a high-performance interconnect
- distributed networks
  - slower interconnect, less tightly synchronized

Next week

How to model parallel programs?

PRAM

Assumptions

- p processors, p large.
- shared memory
- program each proc separately

How to model parallel programs?

PRAM

Assumptions

- p processors, p large.
- shared memory
- program each proc separately

Example problem: AllZeros

- Given array A[1..n].
- Return **true** if A[j] = 0 for all j.
- Return **false** otherwise.

How to model parallel programs?

AllZero(A, 1, n, p)$_j$

for $i = (n/p)(j-1)+1$ **to** $(n/p)(j)$ **do**

  if $A[i] \neq 0$ **then** *answer* = false

done = *done* + 1

wait until (*done* == p)

return *answer*.

How to model parallel programs?

specifies behavior on processor j

$AllZero(A, 1, n, p)_j$

processor j is assigned a specific range of values to examine

for $i = (n/p)(j-1)+1$ **to** $(n/p)(j)$ **do**

  if $A[i] \neq 0$ **then** *answer* = false

done = *done* + 1

wait until (*done* == p)

return *answer*.

How to model parallel programs?

specifies behavior on processor j

$\text{AllZero}(A, 1, n, p)_j$

for $i = (n/p)(j-1)+1$ **to** $(n/p)(j)$ **do**

  if $A[i] \neq 0$ **then** $\textit{answer} = \text{false}$

someone initialized answer in the beginning to true?

$\textit{done} = \textit{done} + 1$

wait until $(\textit{done} == p)$

return *answer*.

How to model parallel programs?

specifies behavior
on processor j

$\text{AllZero}(A, 1, n, p)_j$

for $i = (n/p)(j-1)+1$ **to** $(n/p)(j)$ **do**

  if $A[i] \neq 0$ **then** *answer* = false

done = *done* + 1

wait until (*done* == p)

return *answer*.

Race condition?
Use a lock?

How to model parallel programs?

specifies behavior on processor j

AllZero(A, 1, n, p)$_j$

for i = (n/p)(j-1)+1 **to** (n/p)(j) **do**

if A[i] ≠ 0 **then** *answer* = false

done = *done* + 1

wait until (*done* == p)

return *answer*.

Synchronize with p other processors

How to model parallel programs?

specifies behavior on processor j

AllZero(A, 1, n, p)$_j$

for i = (n/p)(j-1)+1 **to** (n/p)(j) **do**

  if A[i] ≠ 0 **then** *answer* = false

done = *done* + 1

wait until (*done* == p)

return *answer*.

Time: $O\left(\frac{n}{p}\right)$

How to model parallel programs?

PRAM

Assumptions

- p processors, p large.
- shared memory
- program each proc separately

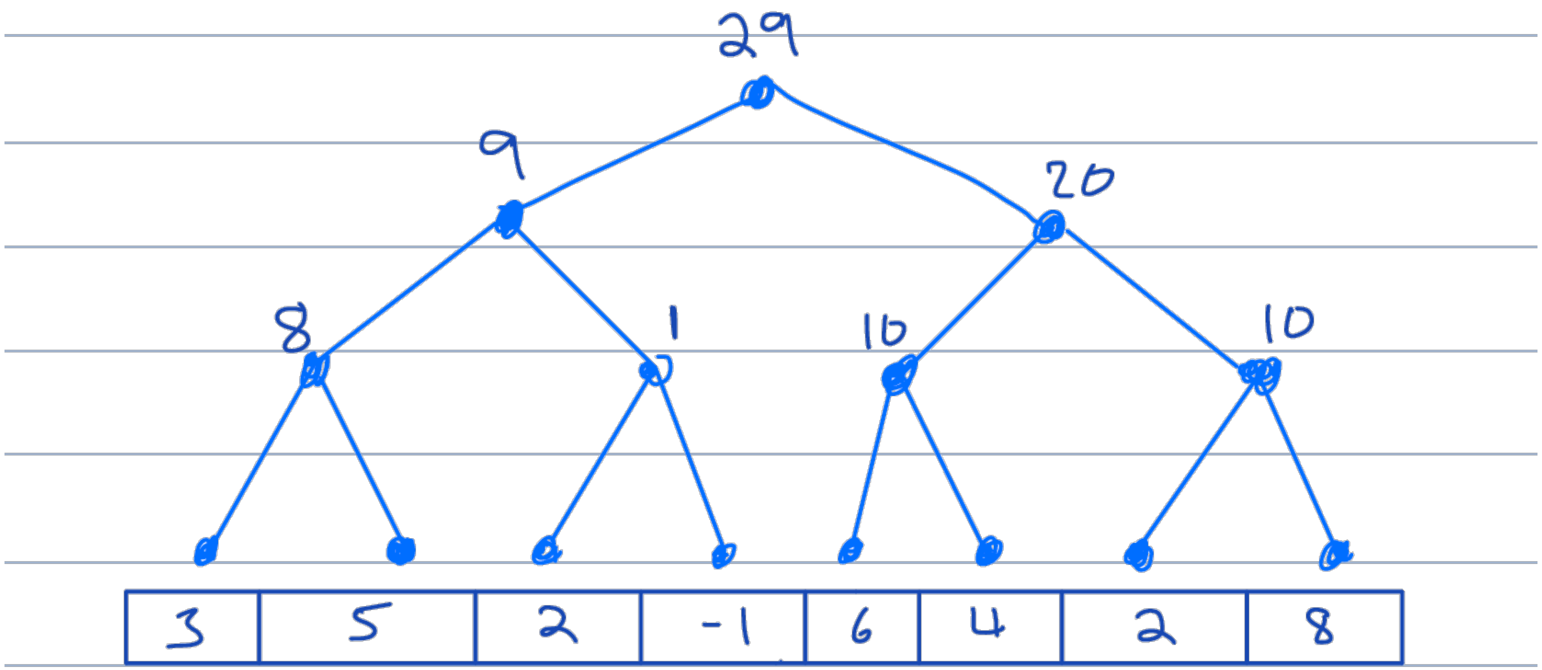
Limitations

- Must carefully manage all processor interactions.
- Manually divide problem among processors.
- Number of processors may be hard-coded into the solution.
- Low-level way to design parallel algorithms.

How to model parallel programs?

Another example: summing an array

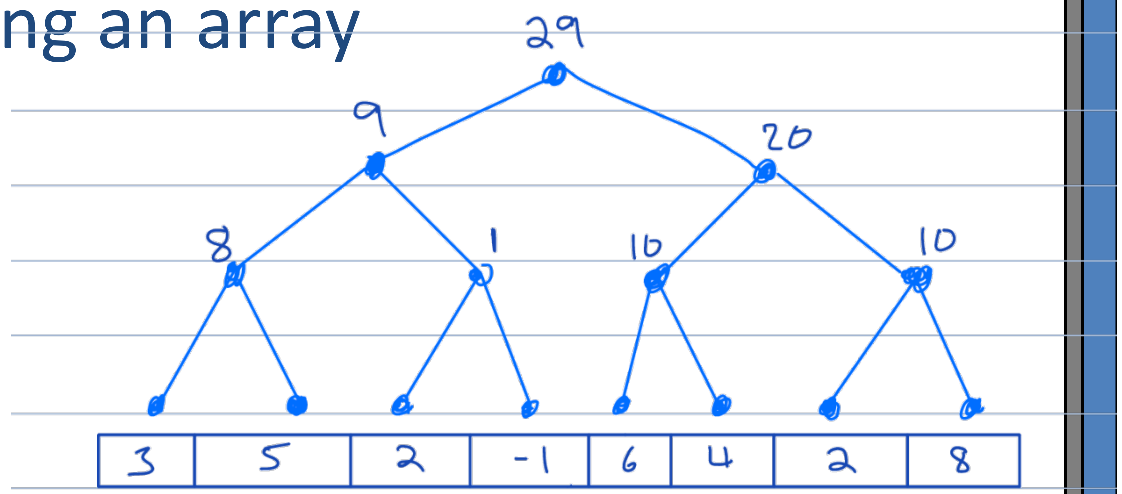
Idea: use a tree



How to model parallel programs?

Another example: summing an array

Algorithm:



RandomSum:

repeat until *root* is not empty:

Choose a random node u in the tree.

If both children are not empty, then:

set u = u.left + u.right

How to model parallel programs?

RandomSum:

repeat until *root* is not empty:

Choose a random node u in the tree.

If both children are not empty, then:

set u = u.left + u.right

Fun exercise: Prove the theorem.

Theorem:

RandomSum finishes in time: $\Theta\left(\frac{n \log n}{p} + \log n\right)$

How to sum an array?

PRAM-Sum:

How to sum an array?

PRAM-Sum:

Assign processors to nodes in tree.

Each processor does assigned work in tree?

Not as easy to specify precise behavior.

How to sum an array?

```
Sum(A[1..n], b, e):
        if (b = e) return A[b]
        mid = (b+e)/2
        in parallel:
        1.   L = Sum(A, b, mid)
        2.   R = Sum(A, mid+2, e)
        sync
        return L+R
```

How to sum an array?

Sum(A[1..n], b, e):

if (b = e) **return** A[b]

mid = (b+e)/2

in parallel:

1. L = Sum(A, b, mid)
2. R = Sum(A, mid+2, e)

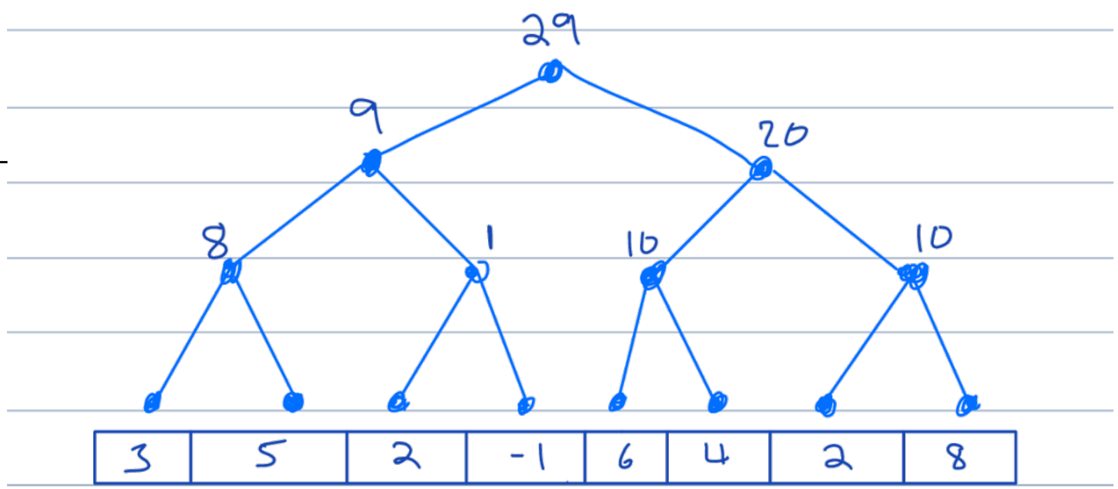
sync

return L+R

Observations:

Same tree calculation!

Each L+R computes 1 node



How to sum an array?

```
Sum(A[1..n], b, e):
        if (b = e) return A[b]
        mid = (b+e)/2
        in parallel:
        1.   L = Sum(A, b, mid)
        2.   R = Sum(A, mid+2, e)
        sync
        return L+R
```

Observations:

Number of processors is not specified anywhere.

How to sum an array?

```
Sum(A[1..n], b, e):
        if (b = e) return A[b]
        mid = (b+e)/2
        in parallel:
        1.  L = Sum(A, b, mid)
        2.  R = Sum(A, mid+2, e)
        sync
        return L+R
```

Observations:

Number of processors is not specified anywhere.

A scheduler assigns parallel computations to processors.

How to sum an array?

```
Sum(A[1..n], b, e):
        if (b = e) return A[b]
        mid = (b+e)/2
        in parallel:
        1.  L = Sum(A, b, mid)
        2.  R = Sum(A, mid+2, e)
        sync
        return L+R
```

Time:

On one processor??

How to sum an array?

```
Sum(A[1..n], b, e):
    if (b = e) return A[b]
    mid = (b+e)/2
    in parallel:
    1.  L = Sum(A, b, mid)
    2.  R = Sum(A, mid+2, e)
    sync
    return L+R
```

Just ignore parallel parts and run all the code!

Time:

On one processor:

$$T_1(n) = 2T_1(n/2) + O(1)$$
$$= O(n)$$

<u>Work</u>
Total steps done by all processors.

How to sum an array?

```
Sum(A[1..n], b, e):
        if (b = e) return A[b]
        mid = (b+e)/2
        in parallel:
        1.  L = Sum(A, b, mid)
        2.  R = Sum(A, mid+2, e)
        sync
        return L+R
```

Time:

On infinite processors??

How to sum an array?

Sum(A[1..n], b, e):

> **if** (b = e) **return** A[b]
>
> *mid* = (b+e)/2
>
> **in parallel:**
>
> 1. L = Sum(A, b, mid)
> 2. R = Sum(A, mid+2, e)
>
> **sync**
>
> **return** L+R

Each parallel part is delegated to two different processors.

Time:

On infinite processors:

Critical Path
or
Span:
longest path in the program

$$T_\infty(n) = T_\infty(n/2) + O(1)$$
$$= O(\log n)$$

How to sum an array?

```
Sum(A[1..n], b, e):
        if (b = e) return A[b]
        mid = (b+e)/2
        in parallel:
        1.   L = Sum(A, b, mid)
        2.   R = Sum(A, mid+2, e)
        sync
        return L+R
```

Time:

On p processors??

$$T_p(n) \quad = \quad ??$$

How to sum an array?

```
Sum(A[1..n], b, e):
        if (b = e) return A[b]
        mid = (b+e)/2
        in parallel:
        1.   L = Sum(A, b, mid)
        2.   R = Sum(A, mid+2, e)
        sync
        return L+R
```

Time:

On p processors??

DEPENDS!

The scheduler matters.

Simple model of parallel computation

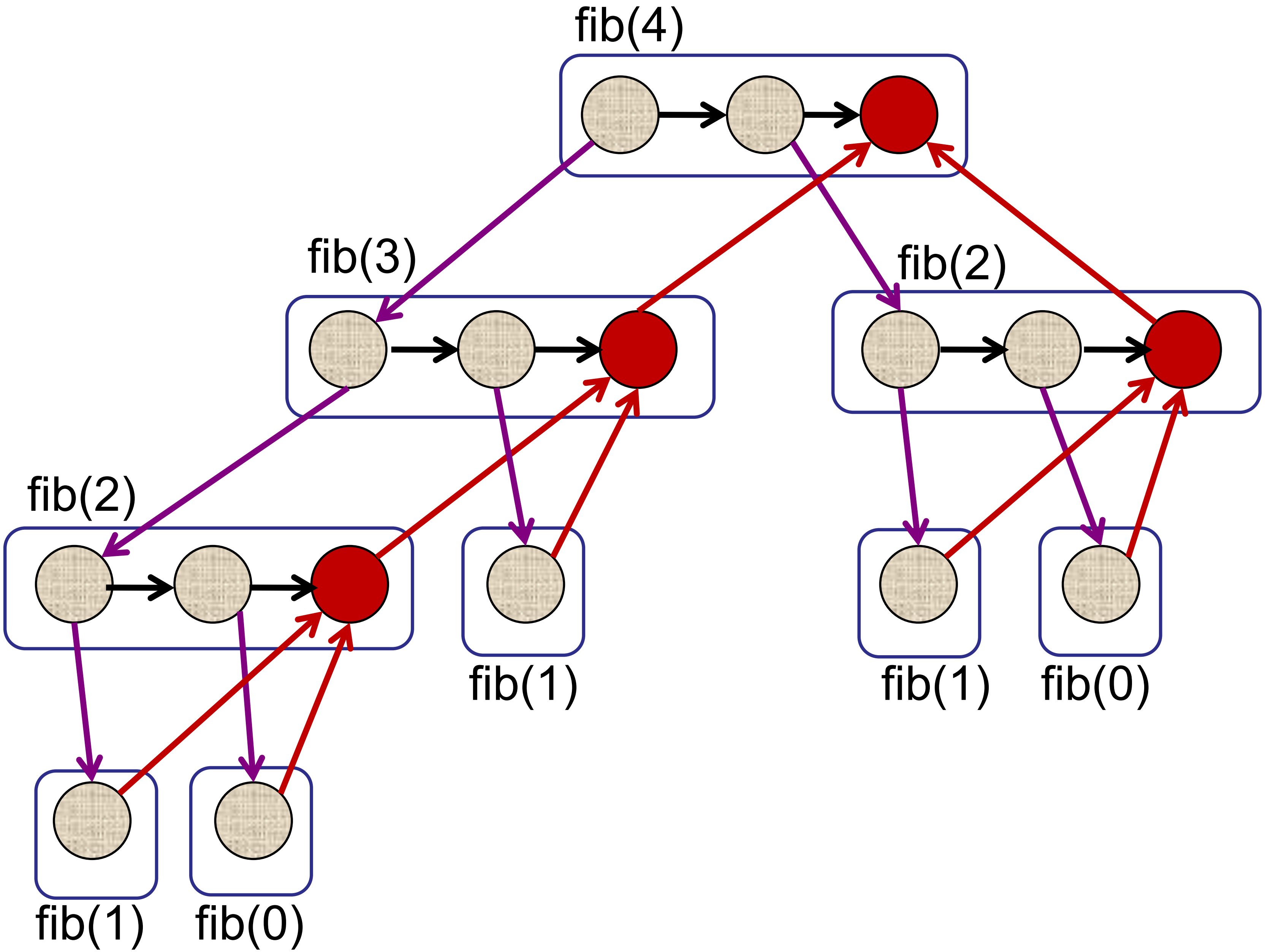
Dynamic Multithreading

- Two special commands:
  - fork (or "in parallel"): start a new (parallel) procedure
  - sync: wait for all concurrent tasks to complete
- Machine independent
  - No fixed number of processors.
- Scheduler assigns tasks to processors.

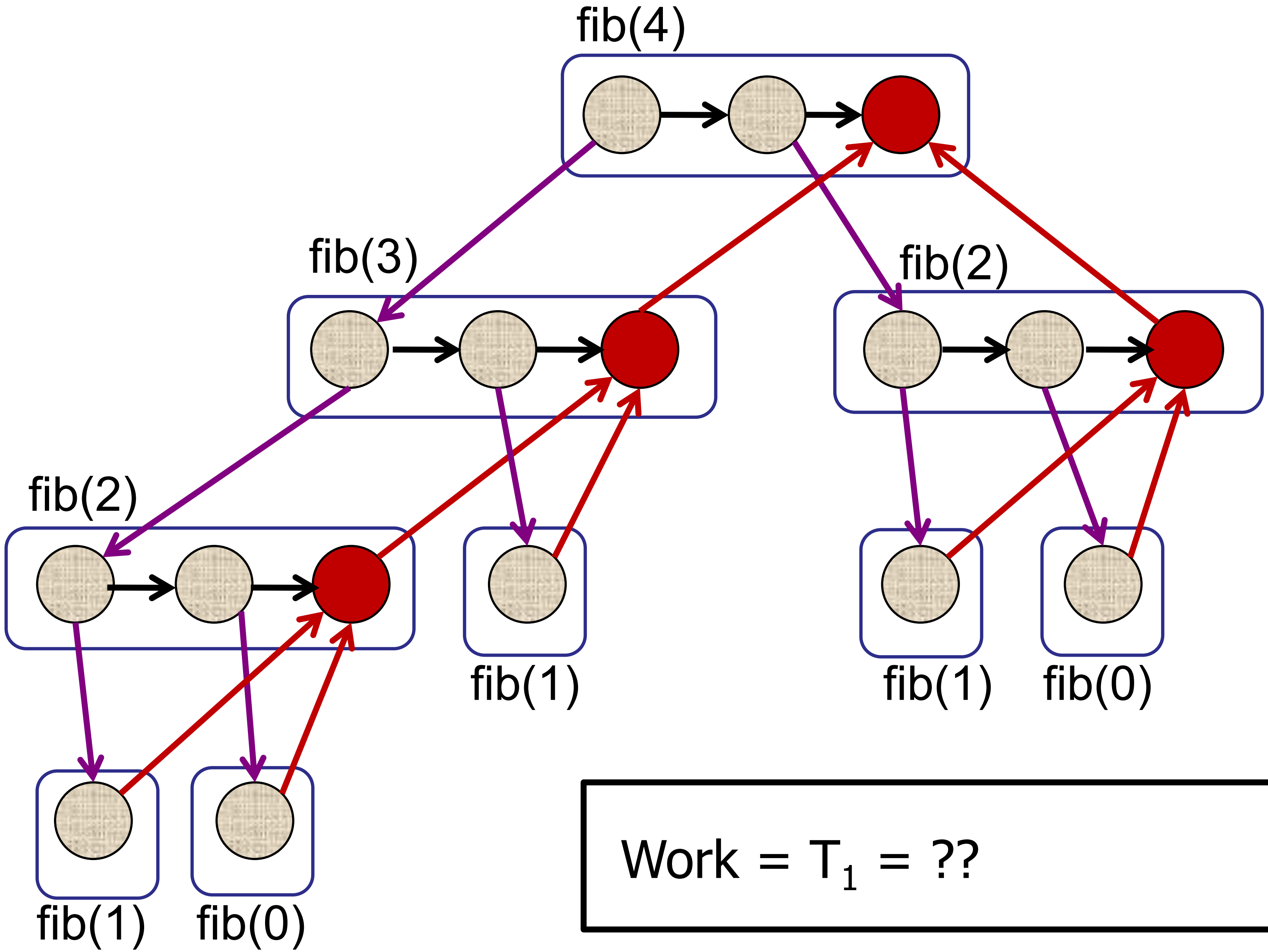
How to sum an array?

```
Sum(A[1..n], b, e):
    if (b = e) return A[b]
    mid = (b+e)/2
    fork:
    1.  L = Sum(A, b, mid)
    2.  R = Sum(A, mid+2, e)
    sync
    return L+R
```

Model as a DAG

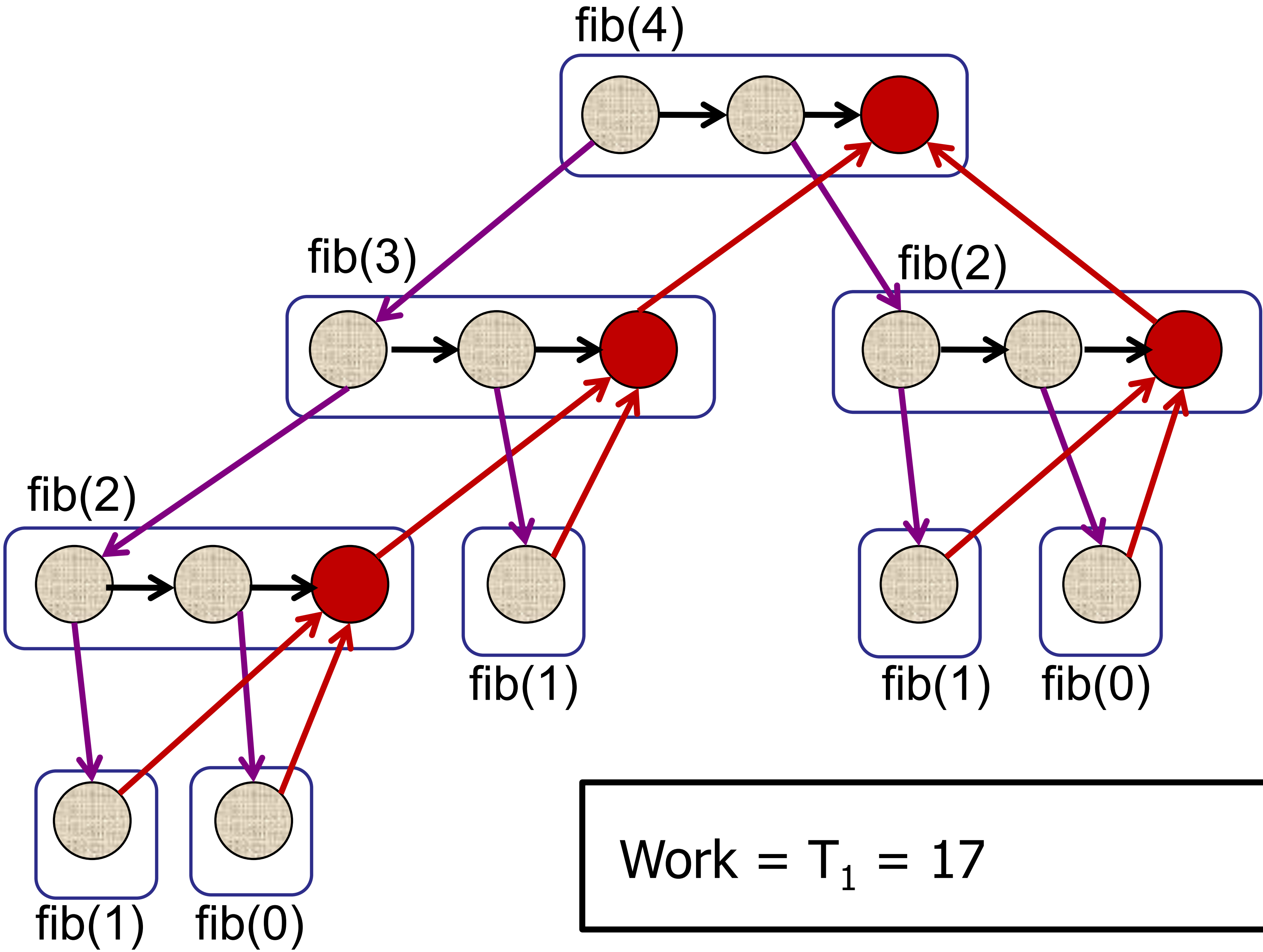


Model as a DAG



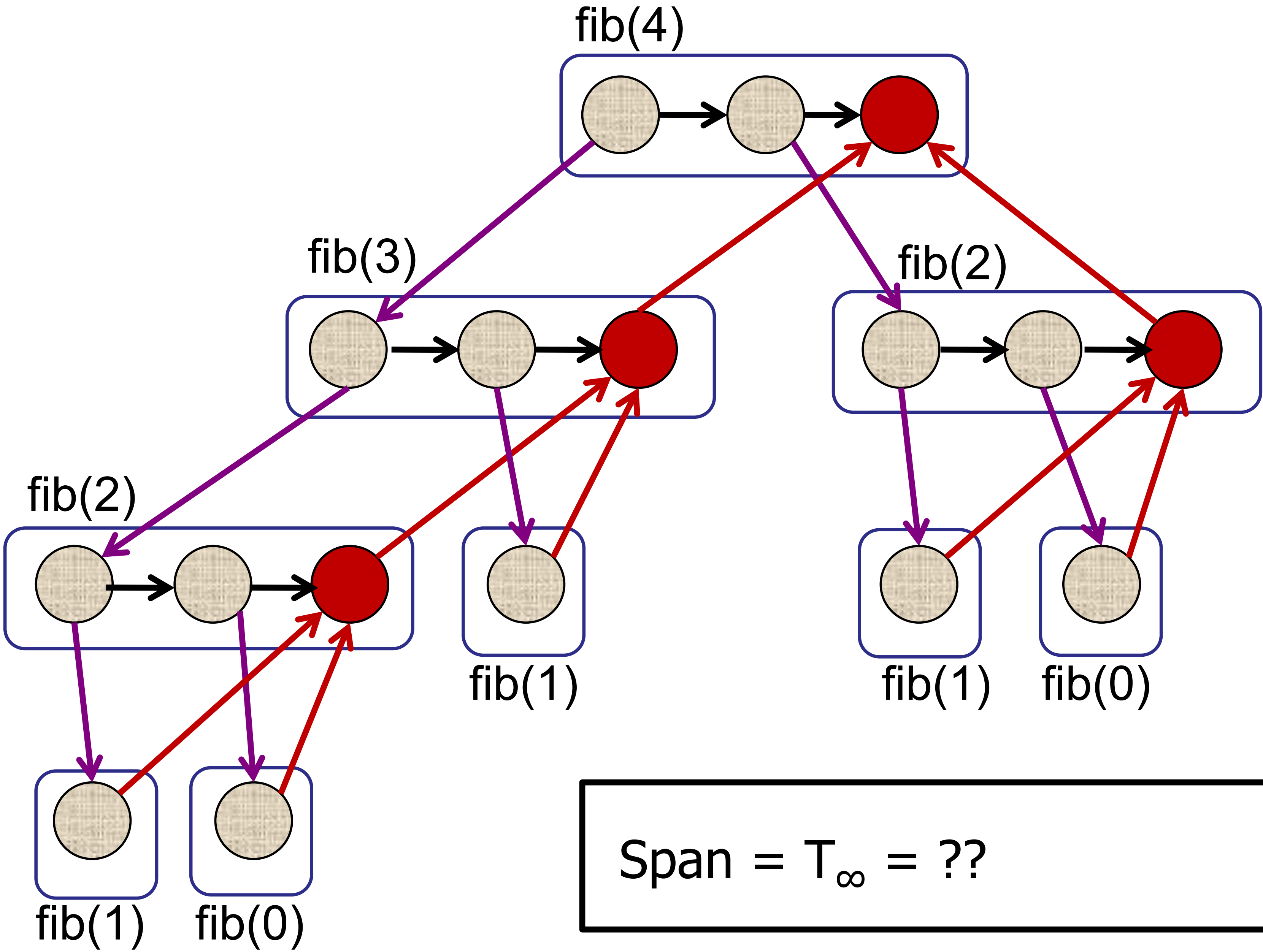
$$\text{Work} = T_1 = ??$$

Model as a DAG



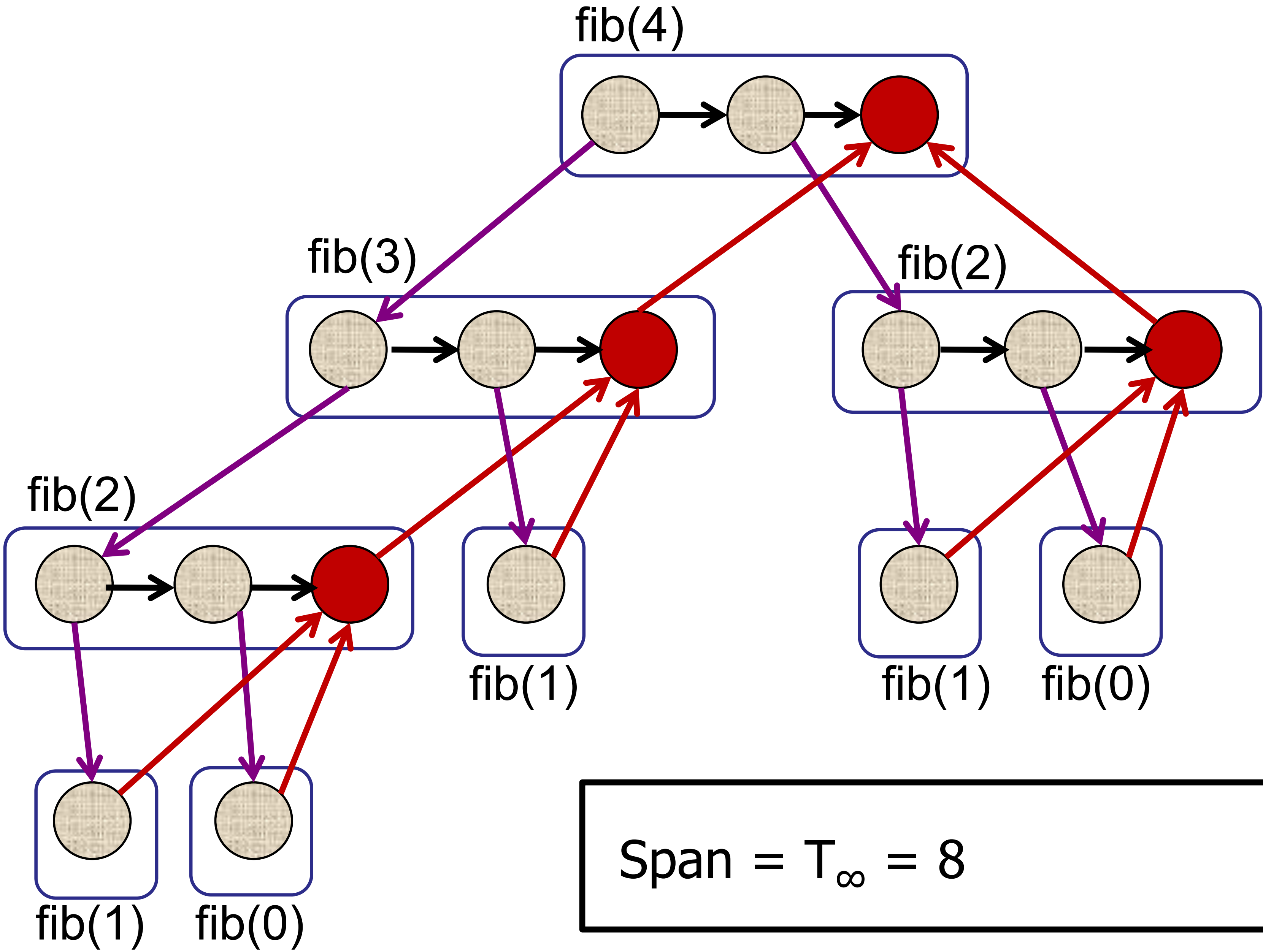
Work = T_1 = 17

Model as a DAG



Span = T_∞ = ??

Model as a DAG

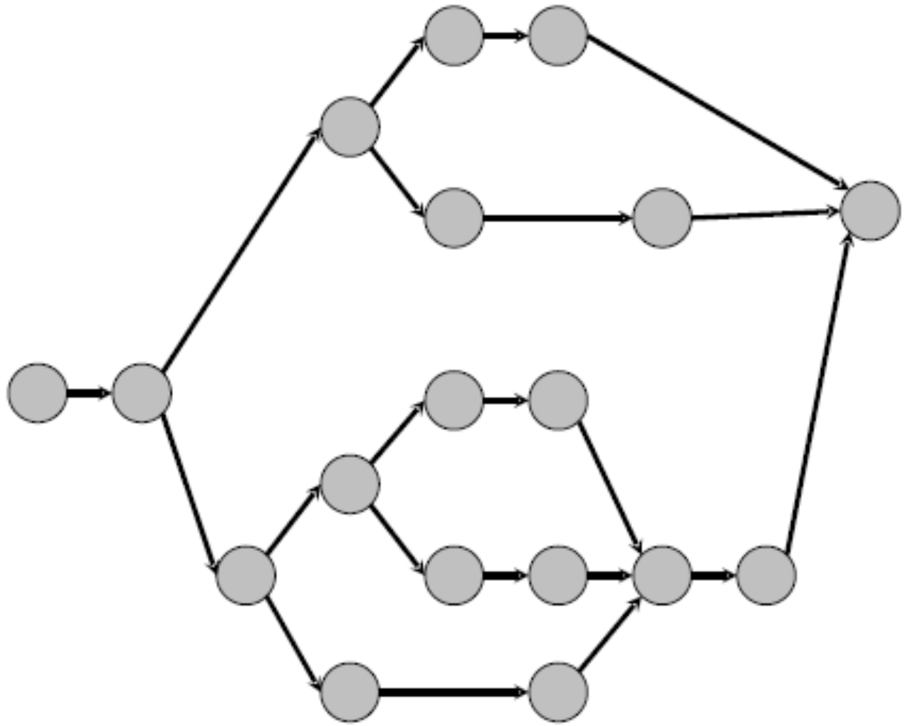


Span = T_∞ = 8

Analyzing Parallel Algorithms

Key metrics:

- Work: T_1
- Span: T_∞

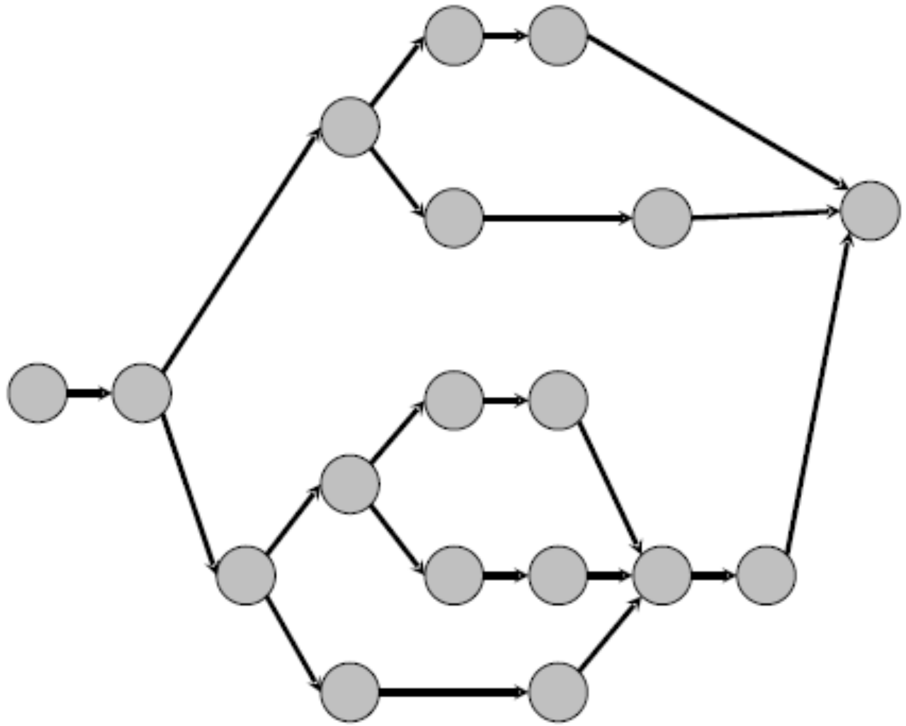


Work = 18
Span = 9

Analyzing Parallel Algorithms

Key metrics:

- Work: T_1
- Span: T_∞



Work = 18
Span = 9
Parallelism = 2

Parallelism:

$$\frac{T_1}{T_\infty}$$

Determines number of processors that we can use productively.

Analyzing a Parallel Computation

Running Time: $\mathbf{T_p}$

- Total running time if executed on ***p*** processors.

- Claim: $\mathbf{T_p > T_\infty}$
  - Cannot run slower on more processors!
  - Mostly, but not always, true in practice.

Analyzing a Parallel Computation

Running Time: $\mathbf{T_p}$

- Total running time if executed on **p** processors.

- Claim: $\mathbf{T_p > T_1 / p}$
  - Total work, divided perfectly evenly over **p** processors.
  - Only for a perfectly parallel program.

Analyzing a Parallel Computation

Running Time: T_p

- Total running time if executed on ***p*** processors.
- $T_p > T_1 / p$
- $T_p > T_\infty$
- Goal: $T_p = (T_1 / p) + T_\infty$
  - Almost optimal (within a factor of 2).
  - We have to spend time T_∞ on the critical path.

    We call this the "sequential" part of the computation.
  - We have to spend time (T_1 / p) doing all the work.

    We call this the "parallel" part of the computation.

Analyzing Parallel Algorithms

Key metrics:

- Work: T_1
- Span: T_∞

Parallelism:

$$\frac{T_1}{T_\infty}$$

Assume p = T_1 / T_∞:

$$\begin{aligned} T_p &= \frac{T_1}{p} + T_\infty \\ &= \frac{T_1}{T_1/T_\infty} + T_\infty \\ &= 2T_\infty \end{aligned}$$

Analyzing a Parallel Computation

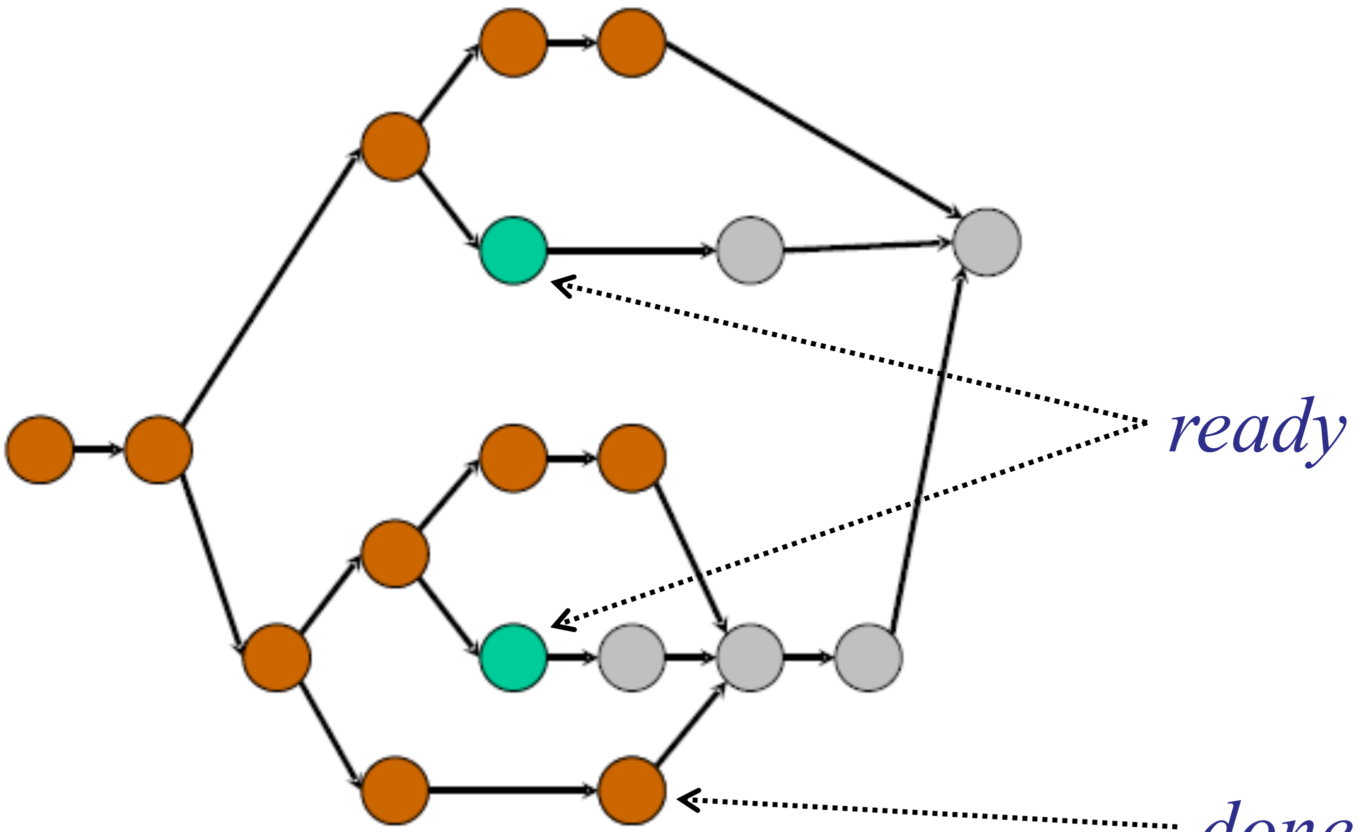
Greedy Scheduler

- If $\leq p$ tasks are *ready*, execute all of them.
- If $> p$ tasks are *ready*, execute p of them.

Analyzing a Parallel Computation

Greedy Scheduler

- If $\leq p$ tasks are *ready*, execute all of them.
- If $> p$ tasks are *ready*, execute p of them.



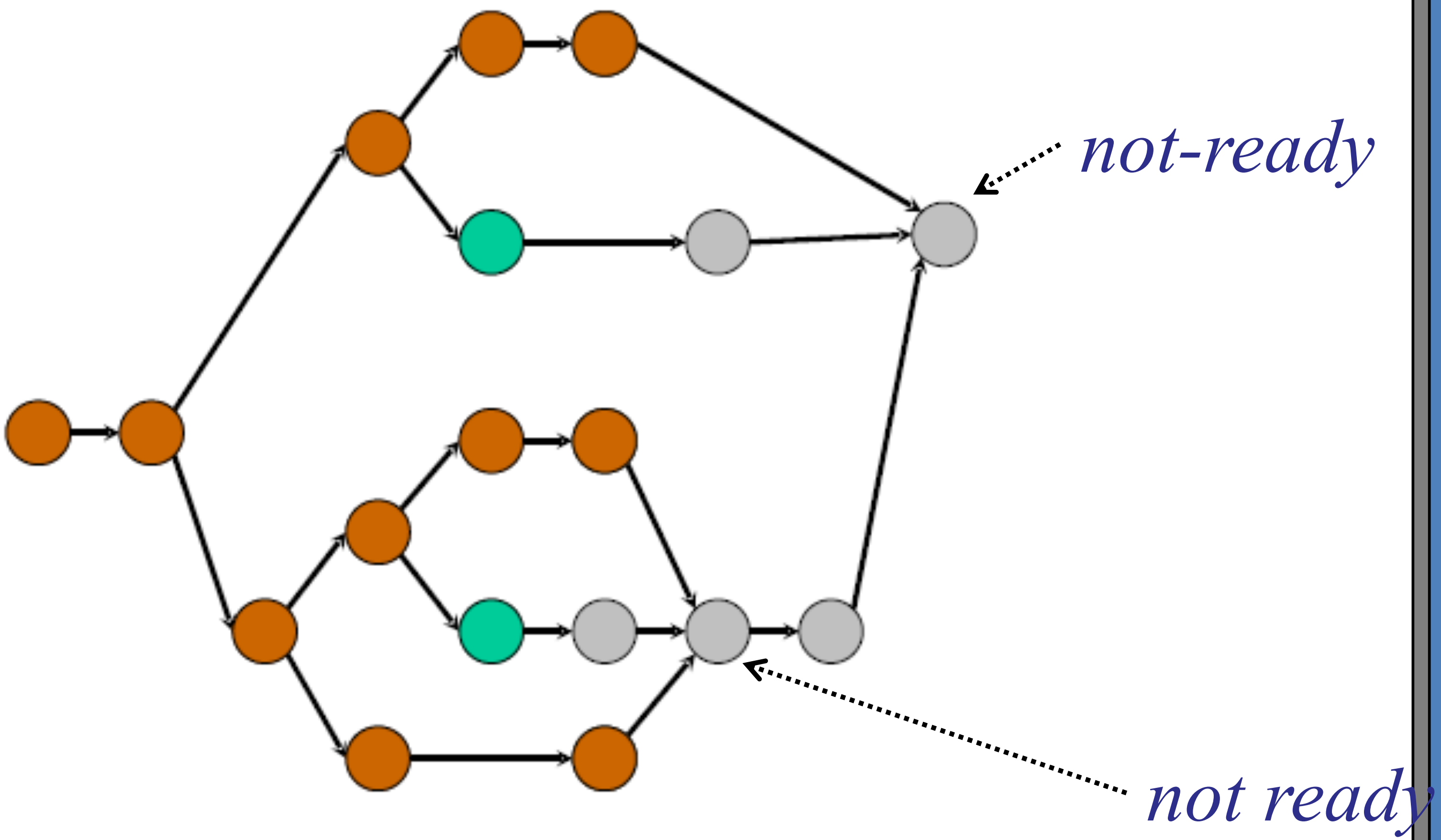
Assume $p = 3$

Analyzing a Parallel Computation

Greedy Scheduler

– If $\leq p$ tasks are *ready*, execute all of them.

– If $> p$ tasks are *ready*, execute p of them.



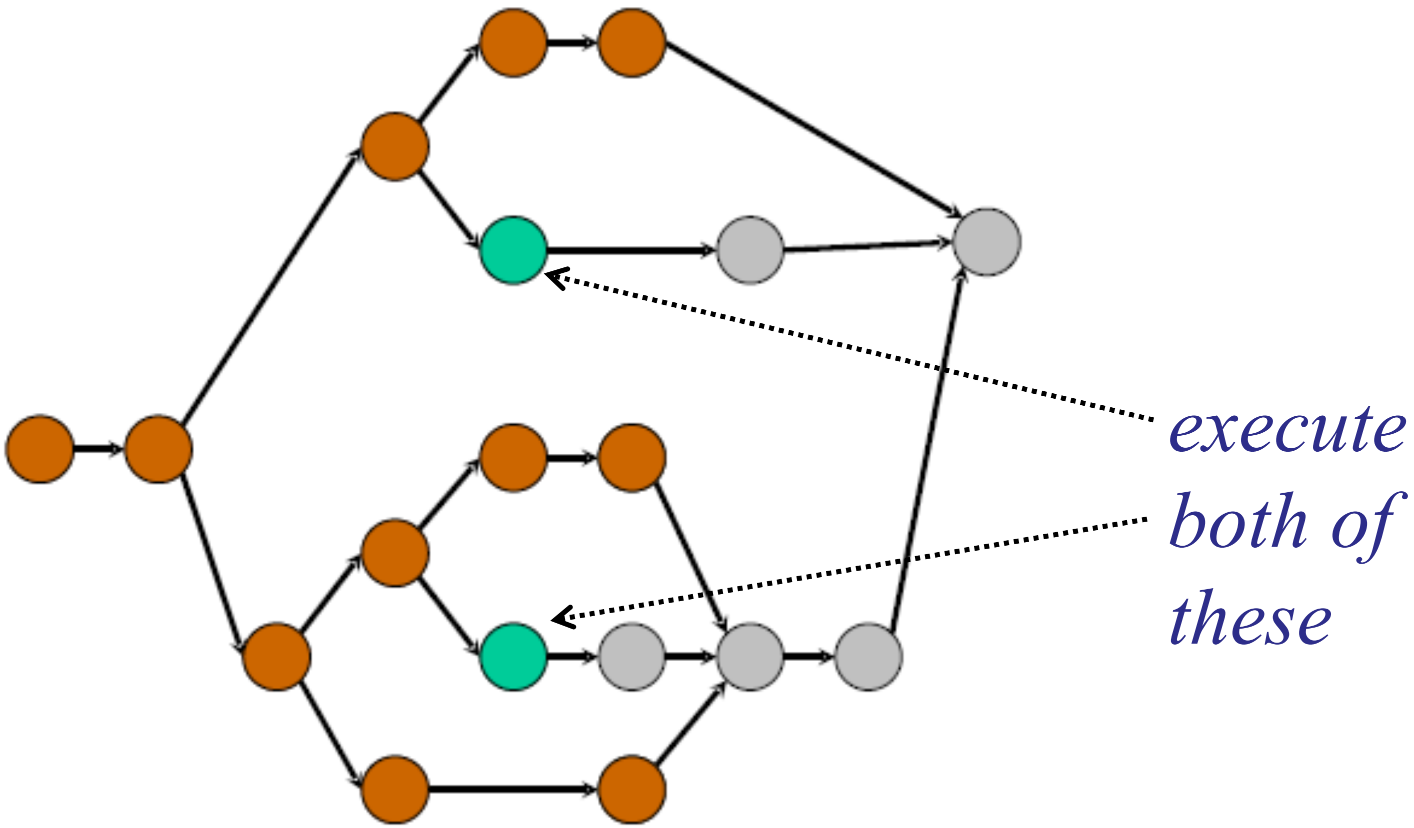
Assume $p = 3$

Analyzing a Parallel Computation

Greedy Scheduler

– If $\leq p$ tasks are *ready*, execute all of them.

– If $> p$ tasks are *ready*, execute p of them.

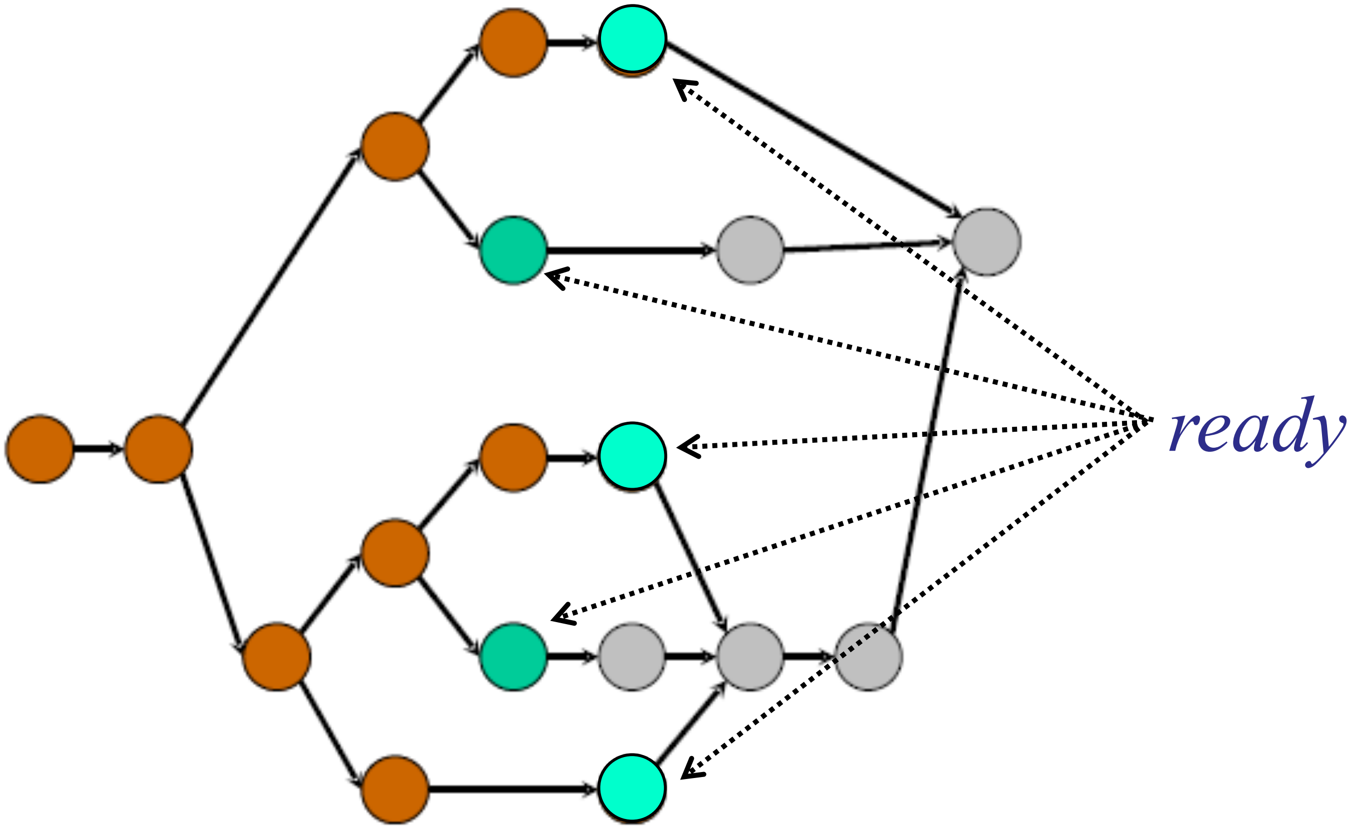


Assume $p = 3$

Analyzing a Parallel Computation

Greedy Scheduler

- If $\leq p$ tasks are *ready*, execute all of them.
- If $> p$ tasks are *ready*, execute p of them.



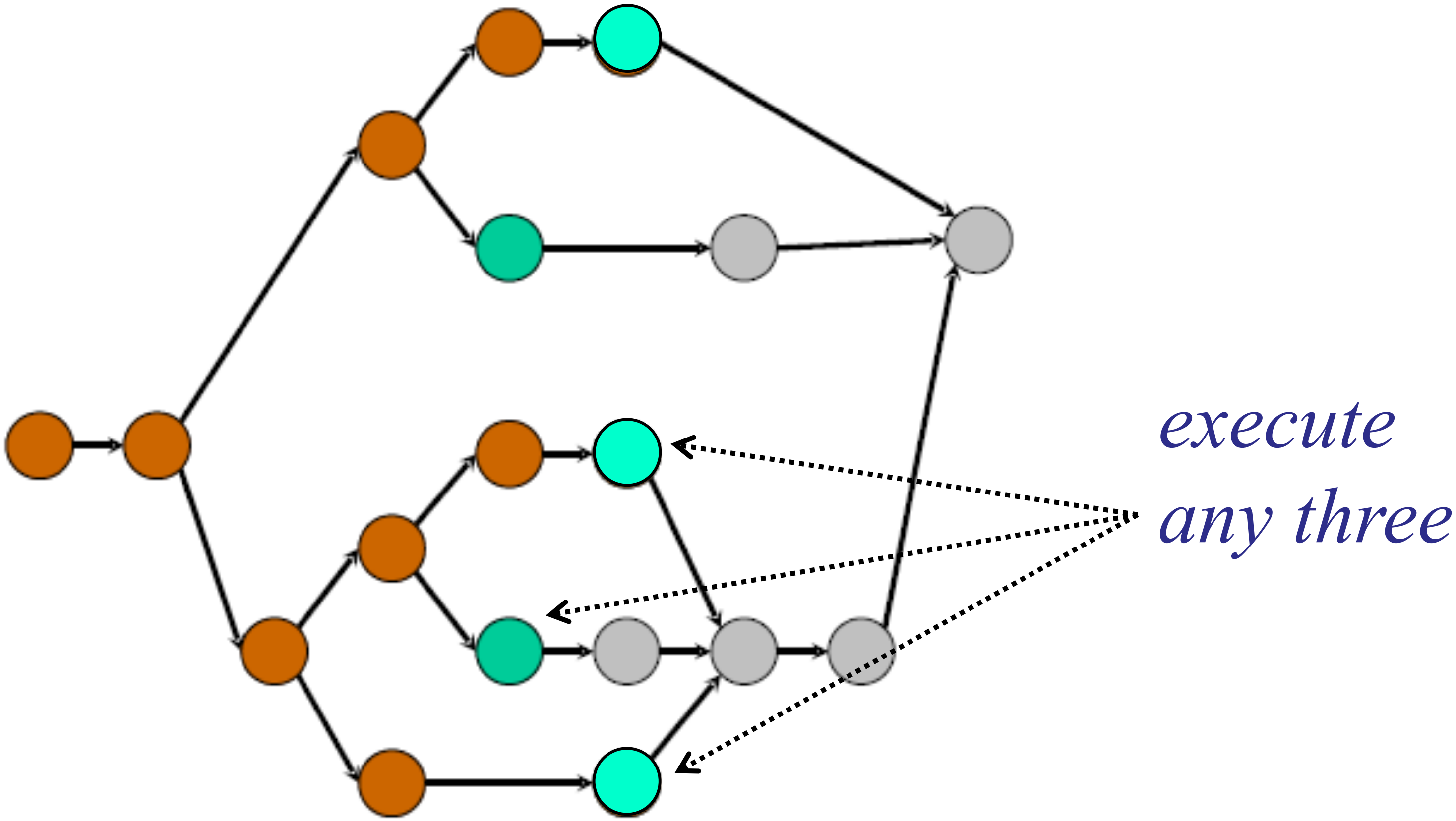
Assume $p = 3$

Analyzing a Parallel Computation

Greedy Scheduler

– If $\leq p$ tasks are *ready*, execute all of them.

– If $> p$ tasks are *ready*, execute p of them.



Assume $p = 3$

Analyzing a Parallel Computation

Greedy Scheduler

1. If $\leq p$ tasks are *ready*, execute all of them.
2. If $> p$ tasks are *ready*, execute p of them.

Theorem (Brent-Graham): $\mathbf{T_p \leq (T_1 / \mathit{p}) + T_\infty}$

Proof:

- At most steps $\mathbf{(T_1 / \mathit{p})}$ of type 2.
- Every step of type 1 works on the critical path, so at most + $\mathbf{T_\infty}$ steps of type 1.

Analyzing a Parallel Computation

Greedy Scheduler

1. If $\leq p$ tasks are *ready*, execute all of them.
2. If $> p$ tasks are *ready*, execute p of them.

Problem:

– Greedy scheduler is *centralized*.

– How to determine which tasks are ready?

– How to assign processors to ready tasks?

Analyzing a Parallel Computation

Work-Stealing Scheduler

- Each process keeps a queue of tasks to work on.
- Each *spawn* adds one task to queue, keeps working.
- Whenever a process is free, it takes a task from a randomly chosen queue (i.e., work-stealing).

Theorem (work-stealing): $\mathbf{T_p \leq (T_1 / \mathit{p}) + O(T_\infty)}$

- See, e.g., Intel Parallel Studio, Cilk, Cilk++, Java, etc.
- Many frameworks exist to schedule parlalel computations.

How to design parallel algorithms

PRAM

- Schedule each processor manually.
- Design algorithm for a specific number of processors.

Fork-Join model

- Focus on parallelism (and think about algorithms).
- Rely on a good scheduler to assign work to processors.

Parallel Sorting

Parallel Sorting

```
MergeSort(A, n)
    if (n=1) then return;
    else
        X = MergeSort(A[1..n/2], n/2)
        Y = MergeSort(A[n/2+1, n], n/2)
        A = Merge(X, Y);
```

Parallel Sorting

```
pMergeSort(A, n)
  if (n==1) then return;
  else
    X = fork pMergeSort(A[1..n/2], n/2)
    Y = fork pMergeSort(A[n/2+1, n], n/2)
    sync;
    A = Merge(X, Y);
```

Parallel Sorting

```
pMergeSort(A, n)
  if (n==1) then return;
  else
    X = fork pMergeSort(A[1..n/2], n/2)
    Y = fork pMergeSort(A[n/2+1, n], n/2)
    sync;
    A = Merge(X, Y);
```

Work Analysis

– $T_1(n) = 2T_1(n/2) + O(n) = O(n \log n)$

Parallel Sorting

```
pMergeSort(A, n)
  if (n==1) then return;
  else
    X = fork pMergeSort(A[1..n/2], n/2)
    Y = fork pMergeSort(A[n/2+1, n], n/2)
    sync;
    A = Merge(X, Y);
```

Critical Path Analysis

– $T_\infty(n) = T_\infty(n/2) + \mathrm{O(n)} = \mathrm{O}(n)$

Oops!

Parallel Merge

How do we merge two arrays A and B in parallel?

Parallel Merge

How do we merge two arrays A and B in parallel?

- Let's try divide and conquer:

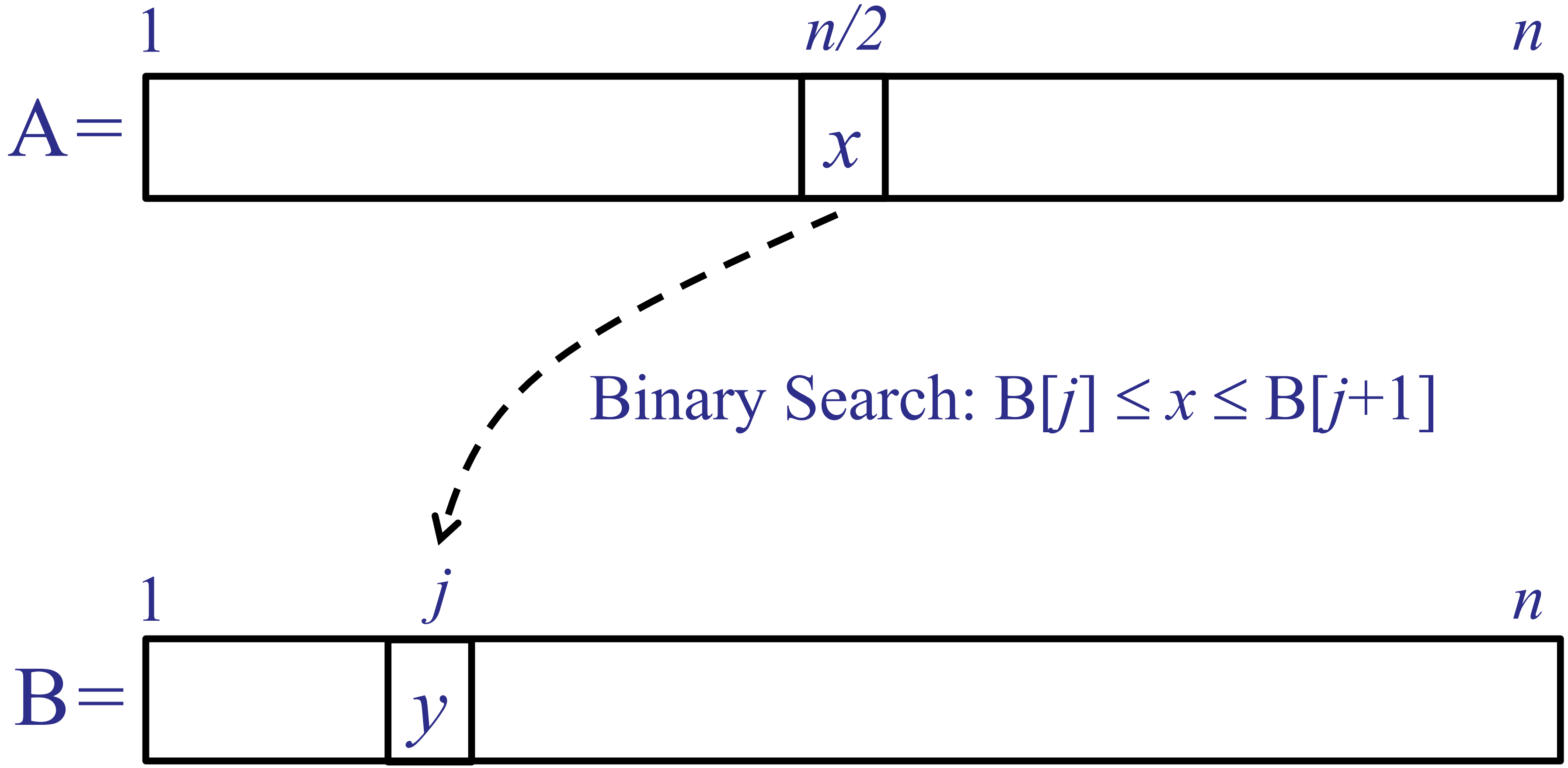
```
X = fork Merge(A[1..n/2], B[1..n/2])
Y = fork Merge(A[n/2+1..n], B[n/2+1..n])
```

A	**=**	**5**	**8**	**9**	**11**	**13**	**20**	**22**	**24**
B	**=**	**6**	**7**	**10**	**23**	**27**	**29**	**32**	**35**

- How do we merge X and Y?

X	**=**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**23**
Y	**=**	**13**	**20**	**22**	**24**	**27**	**29**	**32**	**35**

Parallel Merge



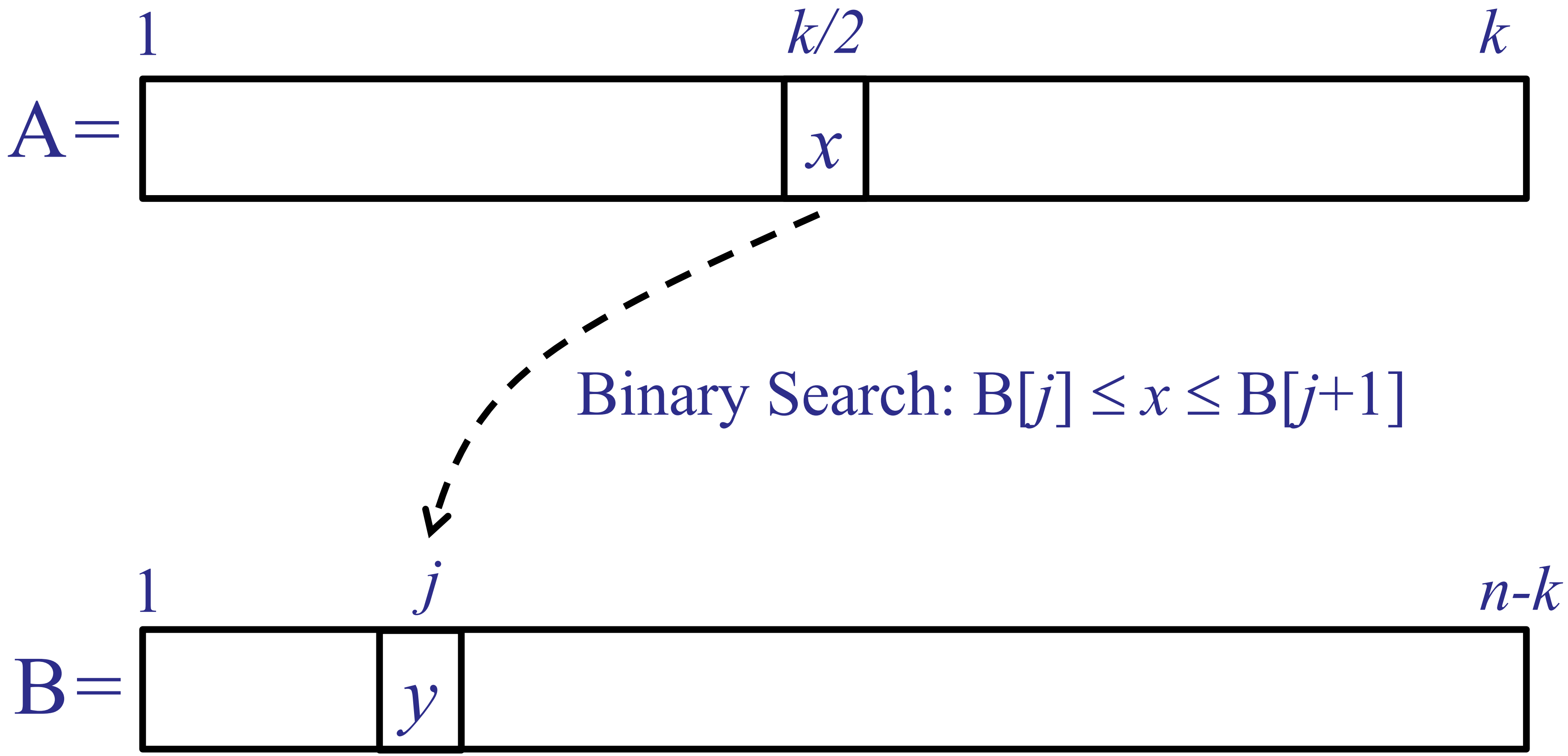
Recurse: **pMerge**(A[1..n/2], B[1..j])

pMerge(A[n/2+1..n], B[j+1..n])

Parallel Merge

```
pMerge(A[1..k], B[1..m], C[1..n])
  if (m > k) then pMerge(B, A, C);
  else if (n==1) then C[1] = A[1];
  else if (k==1) and (m==1) then
    if (A[1] ≤ B[1]) then
      C[1] = A[1]; C[2] = B[1];
    else
      C[1] = B[1]; C[2] = A[1];
  else
    binary search for j where B[j] ≤ A[k/2] ≤ B[j+1]
    fork pMerge(A[1..k/2],B[1..j],C[1..k/2+j])
    fork pMerge(A[k/2+1..l],B[j+1..m],C[k/2+j+1..n])
    sync;
```

Parallel Merge



Recurse: **pMerge**(A[1..n/2], B[1..j])
pMerge(A[n/2+1..n], B[j+1..n])

Parallel Merge

Critical Path Analysis:

- Define $T_\infty(n)$ to be the critical path of parallel merge when the two input arrays A and B together have n elements.
- There are $k > n/2$ elements in A, and $(n\text{-}k)$ elements in B, so in total:
- $k/2 + (n - k) = n - (k/2) < n - (n/4) < 3n/4$
- $T_\infty(n) \leq T_\infty(3n/4) + O(\log n)$

  $\approx O(\log^2 n)$

Parallel Merge

Work Analysis:

- Define $T_1(n)$ to be the work done by parallel merge when the two input arrays A and B together have n elements.
- Fix: $\frac{1}{4} \le \alpha \le \frac{3}{4}$
- $$\begin{aligned} T_1(n) &= T_1(\alpha n) + T_1((1-\alpha)n) + O(\log n) \\ &\approx 2T_1(n/2) + O(\log n) \\ &= O(n) \end{aligned}$$

Parallel Sorting

```
pMergeSort(A, n)
  if (n=1) then return;
  else
    X = fork pMergeSort(A[1..n/2], n/2)
    Y = fork pMergeSort(A[n/2+1, n], n/2)
    sync;
    A = fork pMerge(X, Y);
   sync;
```

Critical Path Analysis

– $T_\infty(n) = T_\infty(n/2) + O(\log^2 n) = O(\log^3 n)$

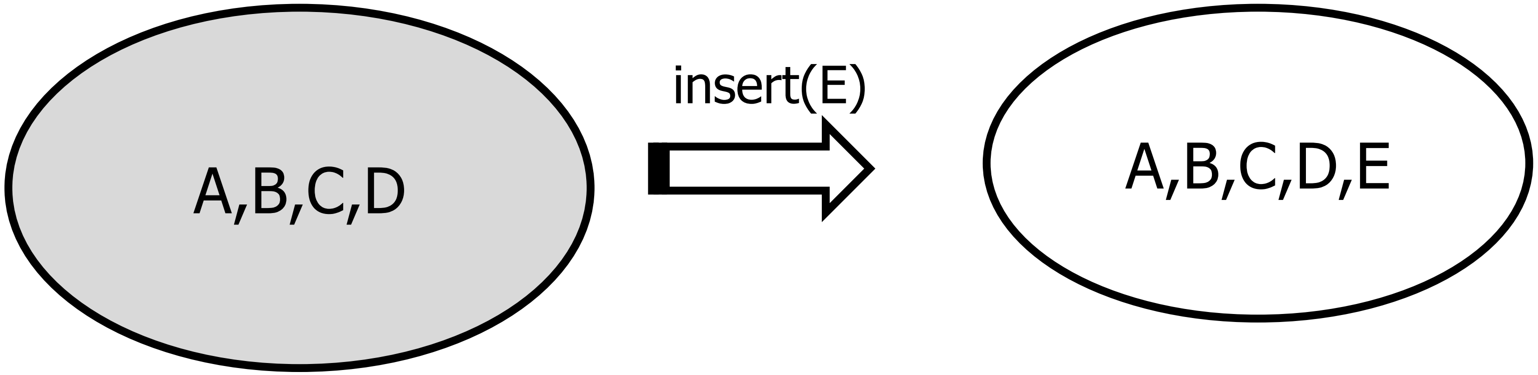
Data Structures

How do we store a set of items?

Data Structures

How do we store a set of items?

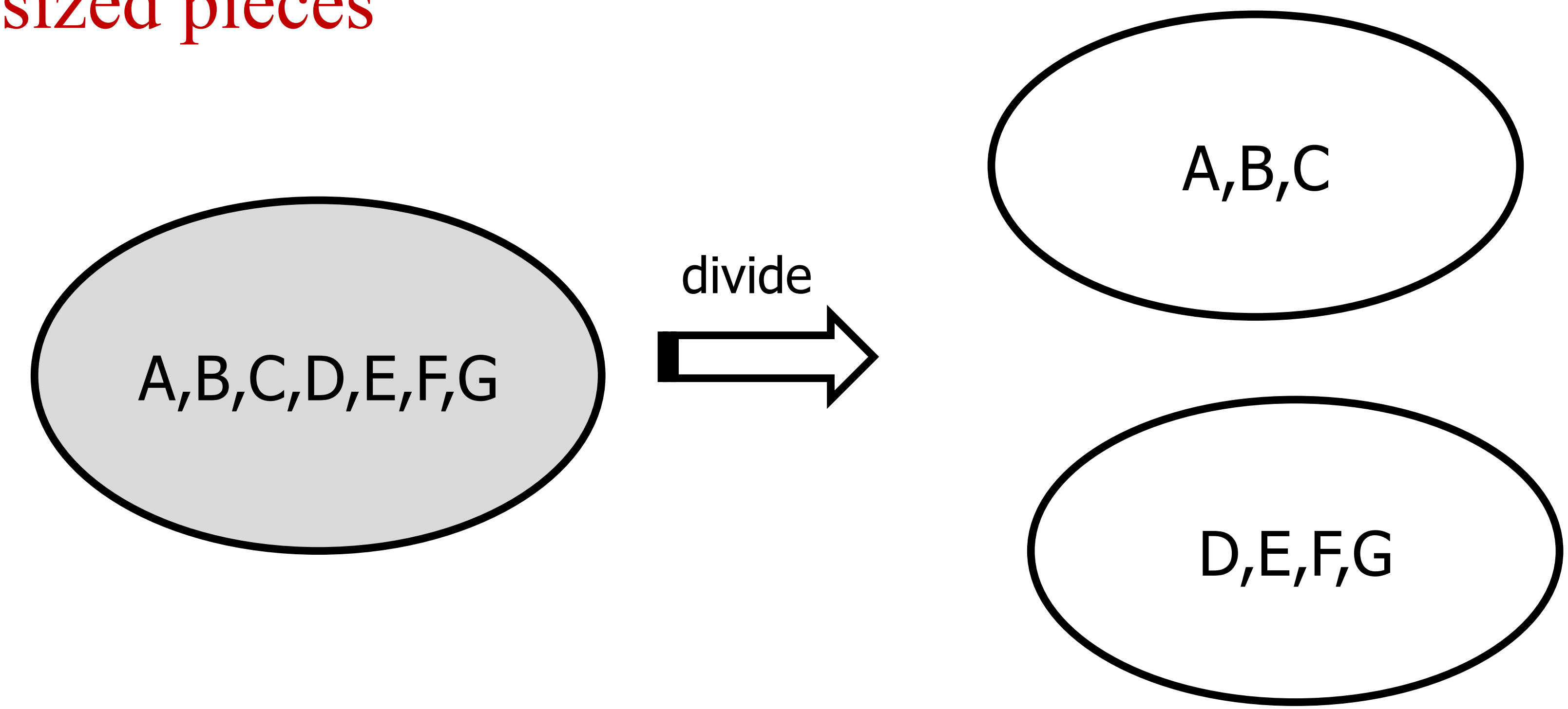
- insert: add an item to the set
- delete: remove an item from the set



Data Structures

How do we store a set of items?

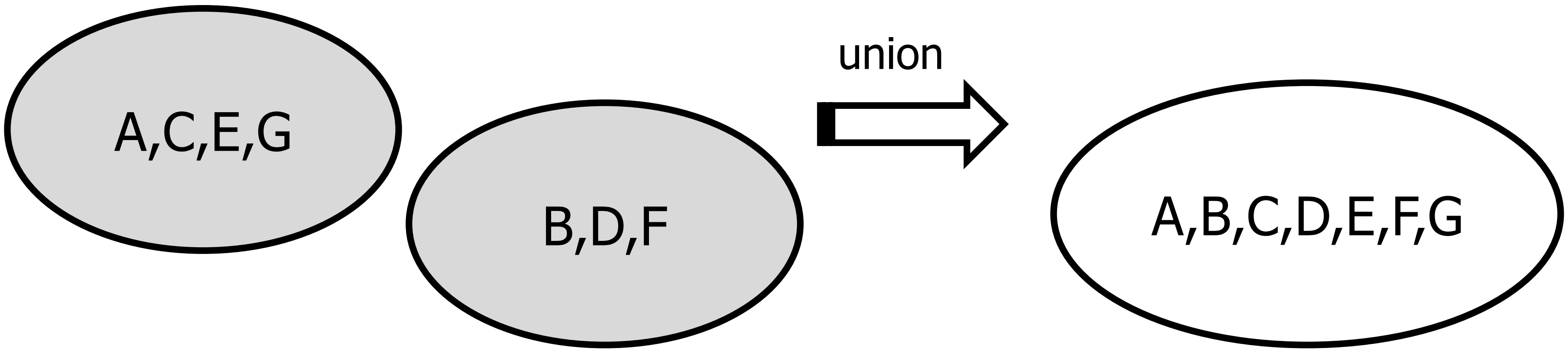
- insert: add an item to the set
- delete: remove an item from the set
- divide: divide the set into two (approximately) equal sized pieces



Data Structures

How do we store a set of items?

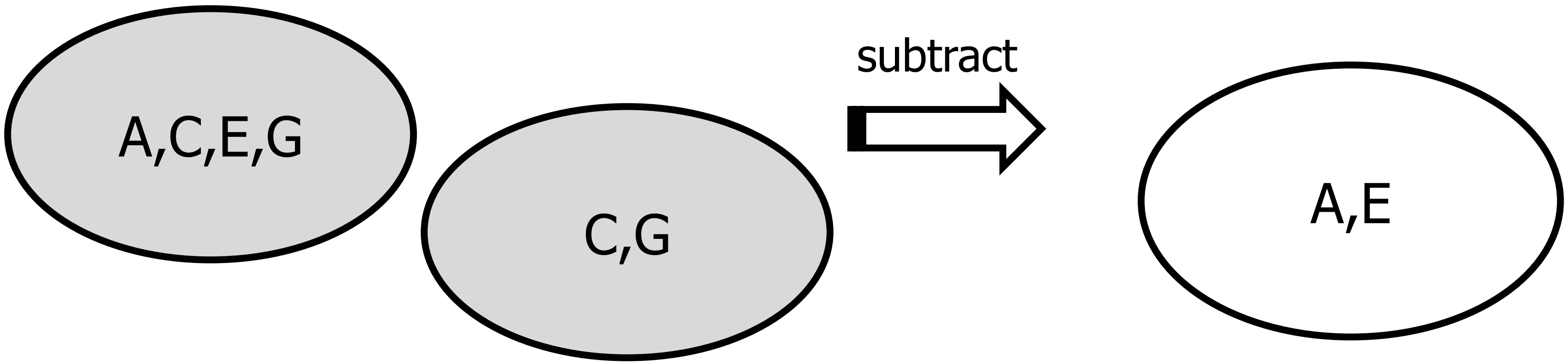
- insert: add an item to the set
- delete: remove an item from the set
- divide: divide the set into two (approximately) equal sized pieces
- union: combine two sets
- subtraction: remove one set from another



Data Structures

How do we store a set of items?

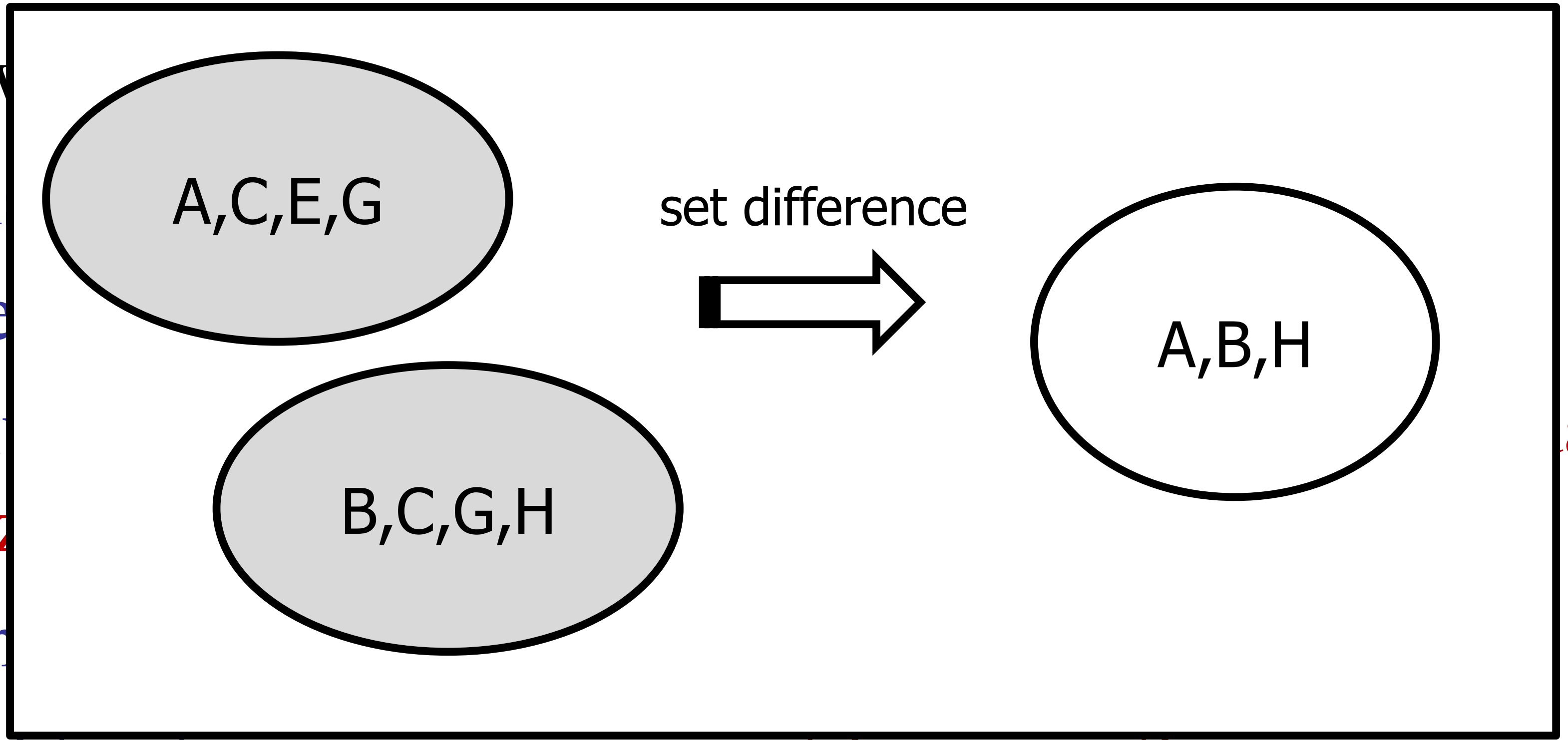
- insert: add an item to the set
- delete: remove an item from the set
- divide: divide the set into two (approximately) equal sized pieces
- union: combine two sets
- subtraction: remove one set from another



Data Structures

Hov

- in
- de
- di ... al
  si
- ur



- subtraction: remove one set from another
- intersection: find the intersection of two sets
- set difference: find the items only in one set

Data Structures

How do we store a set of items?

- insert: add an item to the set
- delete: remove an item from the set
- divide: divide the set into two (approximately) equal sized pieces
- union: combine two sets
- subtraction: remove one set from another
- intersection: find the intersection of two sets
- set difference: find the items only in one set

Data Structures

How do we store a set of items?

- insert: add an item to the set
- delete: remove an item from the set
- divide: divide the set into two (approximately) equal sized pieces

Cost:

n items → $T_1 = O(\log n)$
$T_\infty = O(\log n)$

Data Structures

How do we store a set of items?

- insert: add an item to the set
- delete: remove an item from the set
- divide: divide the set into two (approximately) equal sized pieces

Cost:

n items ➔ $T_1 = O(\log n)$
$T_\infty = O(\log n)$

Sequential solution:
Any balanced binary search tree.

Data Structures

Cost: set 1 (n items), set 2 (m items), $n > m$

➔ $T_1 = O(n + m)$
$T_\infty = O(\log n + \log m)$

- union: combine two sets
- subtraction: remove one set from another
- intersection: find the intersection of two sets
- set difference: find the items only in one set

Data Structures

Cost: set 1 (n items), set 2 (m items), $n > m$

→ $T_1 = O(n + m)$ ← need linear time to examine all items in both sets!

$T_\infty = O(\log n + \log m)$

- union: combine two sets
- subtraction: remove one set from another
- intersection: find the intersection of two sets
- set difference: find the items only in one set

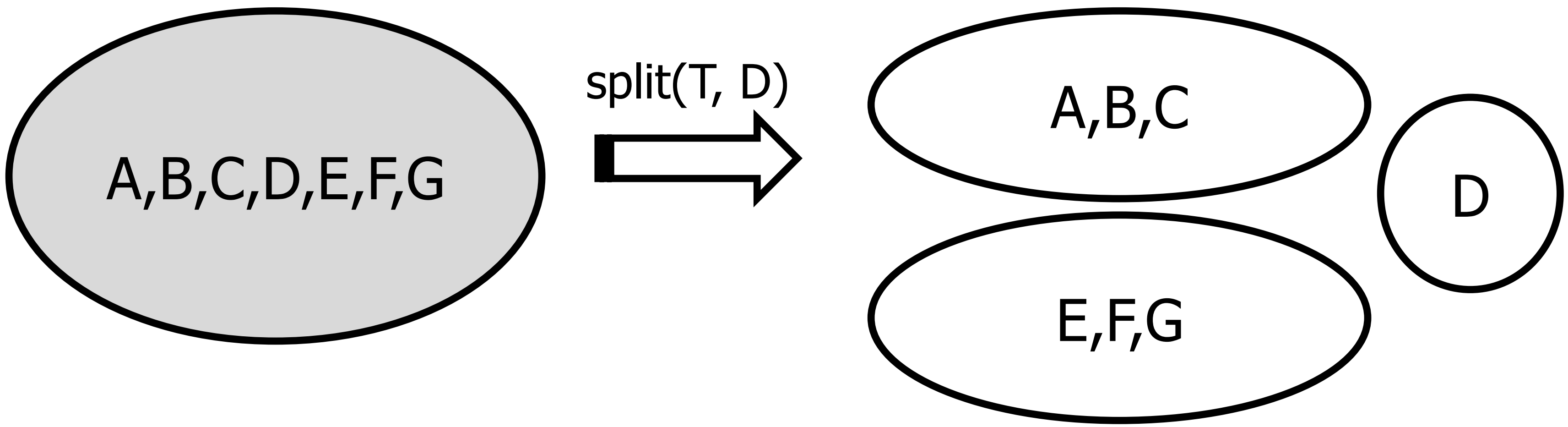
Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

1. split(T, k) ➔ (T1, T2, x)

T1 contains all items < k
T2 contains all items > k
x = k if k was in T



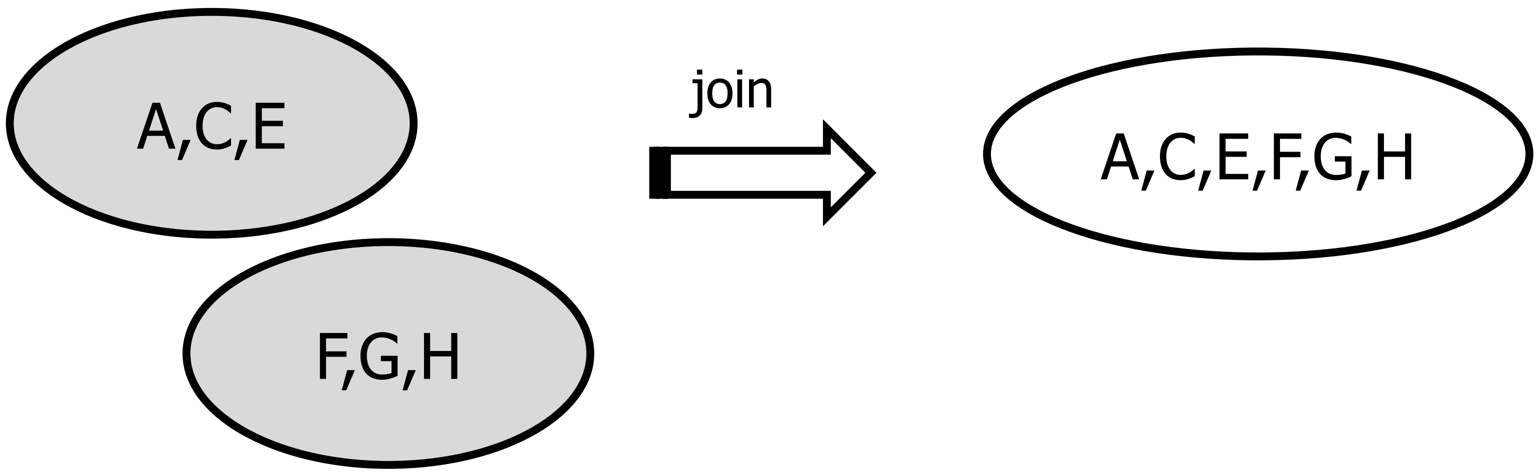
Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

2. join(T1, T2) ➔ T

every item in T1 < every item in T2



Parallel Sets

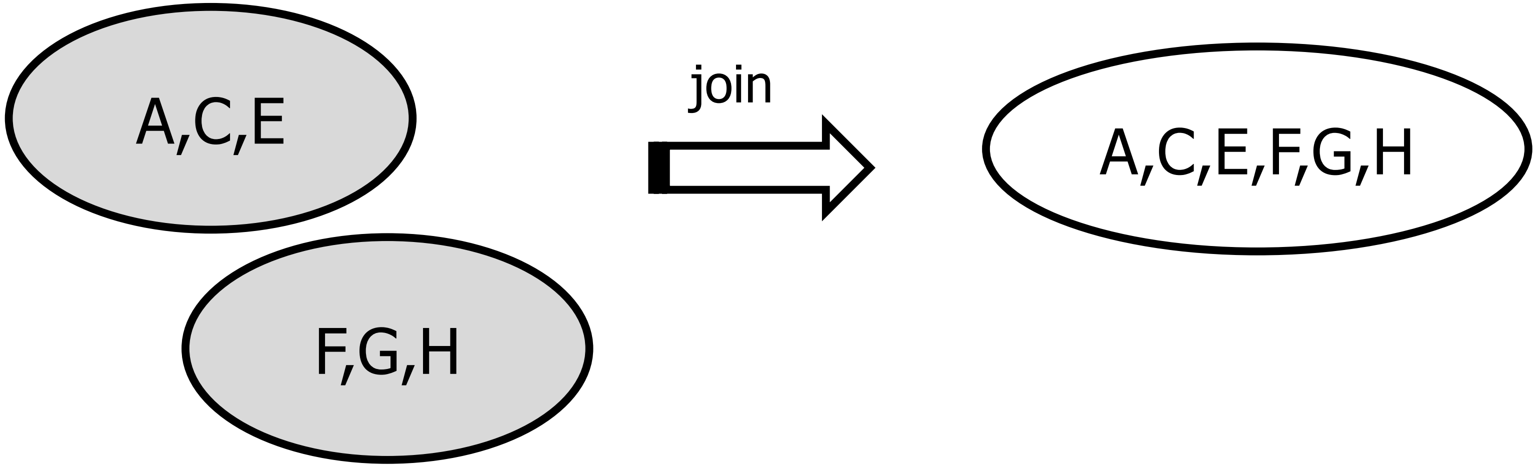
Basic building block:

Note: easier than Union operation because trees are ordered and disjoint!

Balanced binary tree that supports four operations:

2. join(T1, T2) ➔ T

every item in T1 < every item in T2



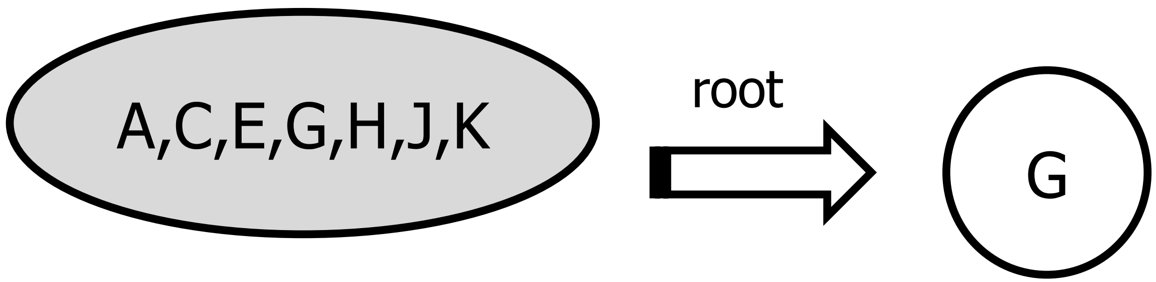
Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

3. root(T) → item at root

Tree T is unchanged.
Root is approximate median.



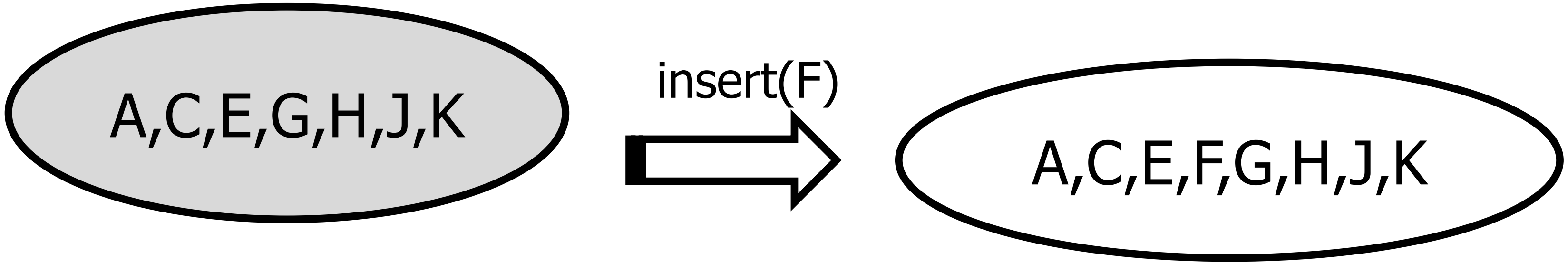
Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

4. insert(T, x) ➔ T'

Tree T' = T with x inserted.



Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

1. split(T, k) ➔ (T1, T2, x)
2. join(T1, T2) ➔ T
3. root(T) ➔ x
4. insert(T, x) ➔ T'

Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

1. split(T, k) ➔ (T1, T2, x)
2. join(T1, T2) ➔ T
3. root(T) ➔ x
4. insert(T, x) ➔ T'

Can implement all four operations with a (2,4)-tree with:

- Work: O(log n + log m)
- Span: O(log n + log m)

Parallel Sets

Basic building block:

Balanced binary tree that supports four operations:

1. split(T, k) → (T1, T2, x)
2. join(T1, T2) → T
3. root(T) → x
4. insert(T, x) → T'

Can implement all four operations with a (2,4)-tree with:

- Work: O(log n + log m)
- Span: O(log n + log m)

Exercise!

Data Structures

How do we store a set of items?

Easy!

- insert: add an item to the set
- delete: remove an item from the set
- divide: divide the set into two (approximately) equal sized pieces
- union: combine two sets
- subtraction: remove one set fr
- intersection: find the intersecti
- set difference: find the items o

Example:

delete(T, k):

(T1, T2, x) = split(T, k)

T = join(T1, T2)

Data Structures

How do we store a set of items?

Easy!

- insert: add an item to the set
- delete: remove an item from the set
- divide: divide the set into two (approximately) equal sized pieces
- union: combine two sets
- subtraction: remove one set fr
- intersection: find the intersecti
- set difference: find the items o

Example:

```
divide(T, k):
   k = root(T)
   (T1, T2, x) = split(T, k)
   T2 = insert(T2, k)
```

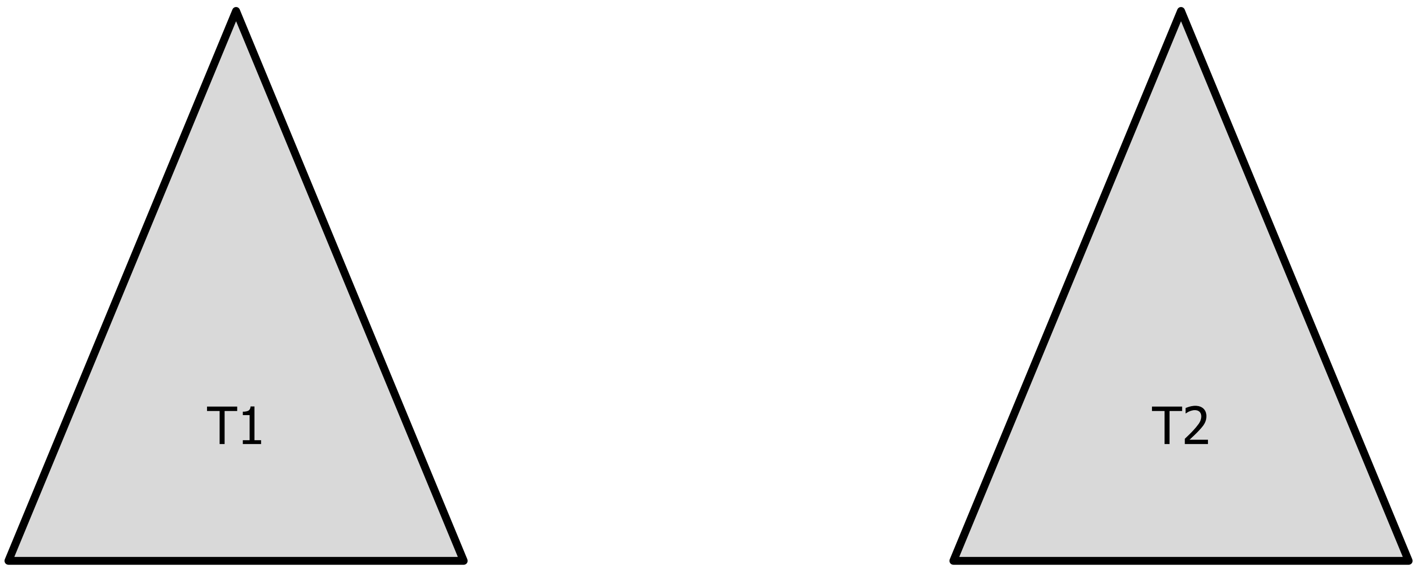
Parallel Sets

Union(T1, T2)

if T1 = null: **return** T2

if T2 = null: **return** T1

…



Parallel Sets

Union(T1, T2)

if T1 = null: **return** T2

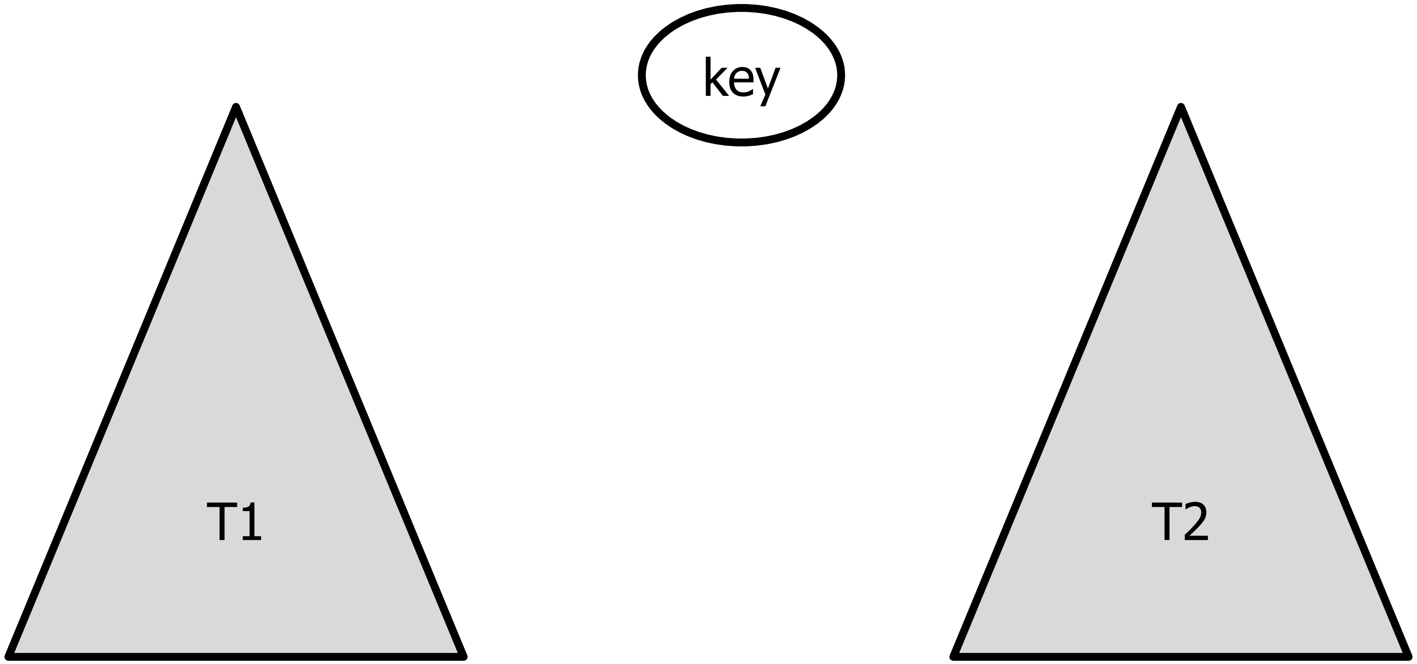
if T2 = null: **return** T1

key = root(T1)

(L, R, x) = split(T2, key)

fork:

…



Parallel Sets

Union(T1, T2)

if T1 = null: **return** T2

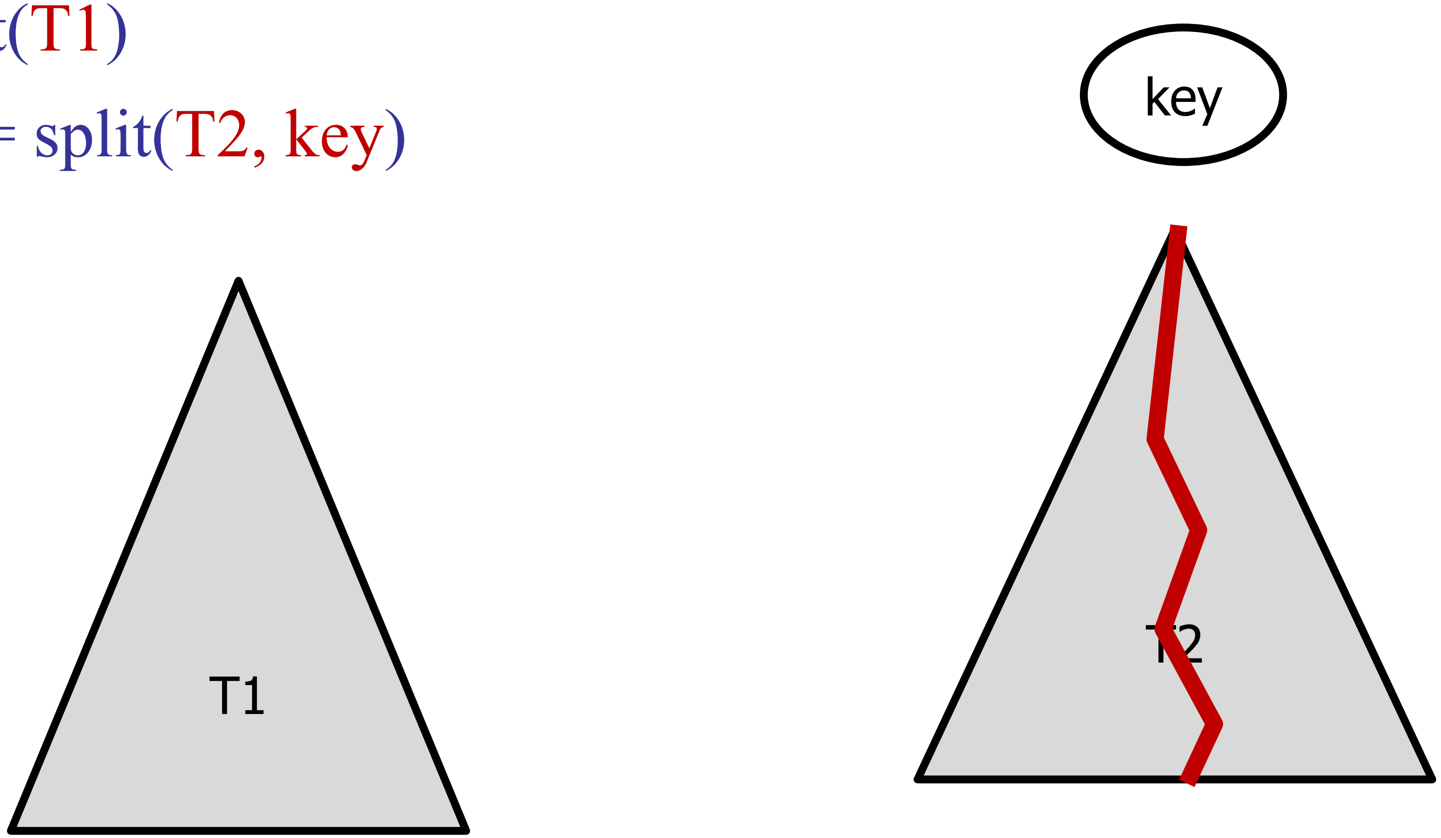
if T2 = null: **return** T1

key = root(T1)

(L, R, x) = split(T2, key)

fork:

…



Parallel Sets

Union(T1, T2)

if T1 = null: **return** T2

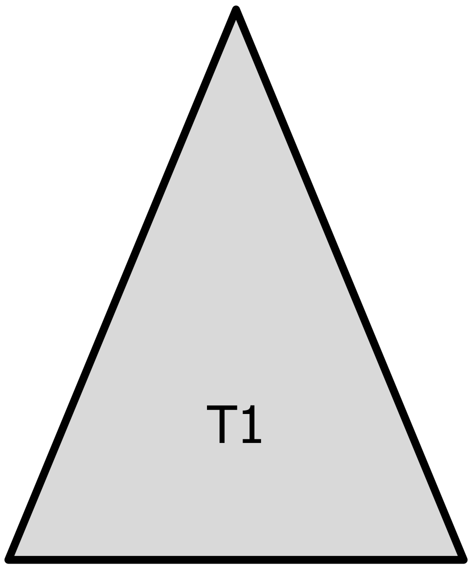
if T2 = null: **return** T1

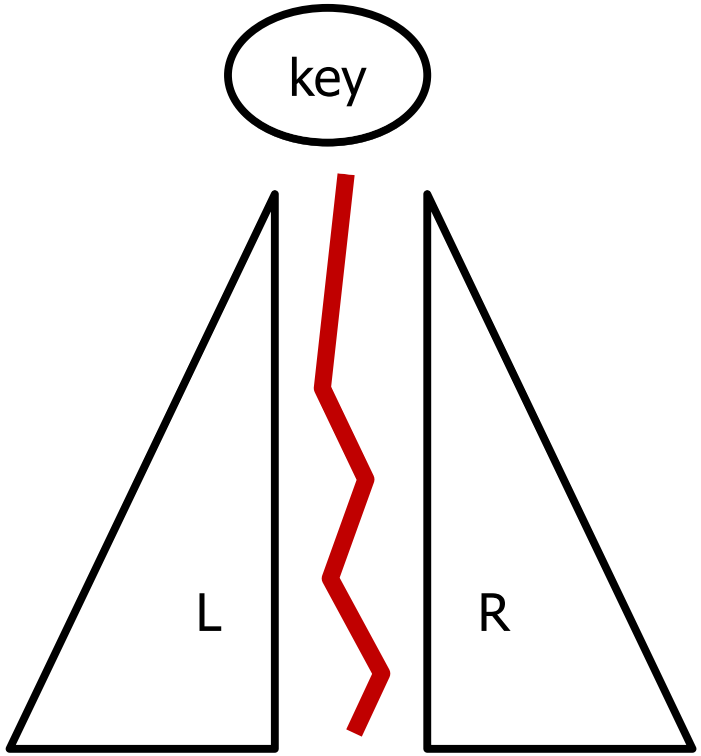
key = root(T1)

(L, R, x) = split(T2, key)

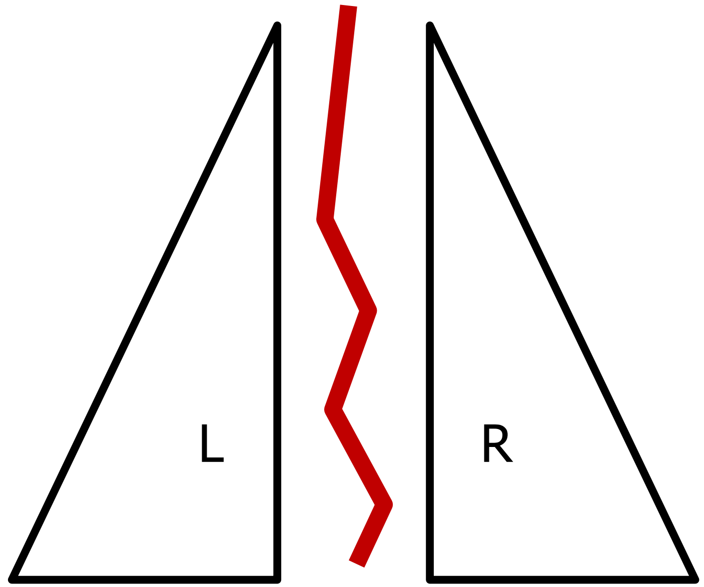
fork:

…



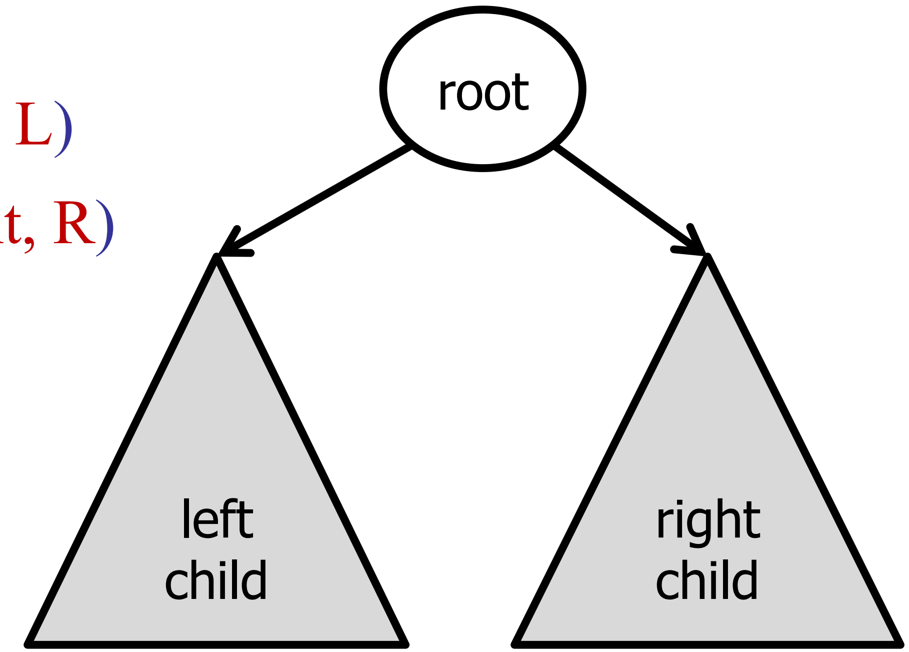


Parallel Sets



Union(T1, T2)

```
if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1.   TL = Union(key.left, L)
2.   TR = Union(key.right, R)
sync
…
```



Parallel Sets

Union(T1, T2)

if T1 = null: **return** T2

if T2 = null: **return** T1

key = root(T1)

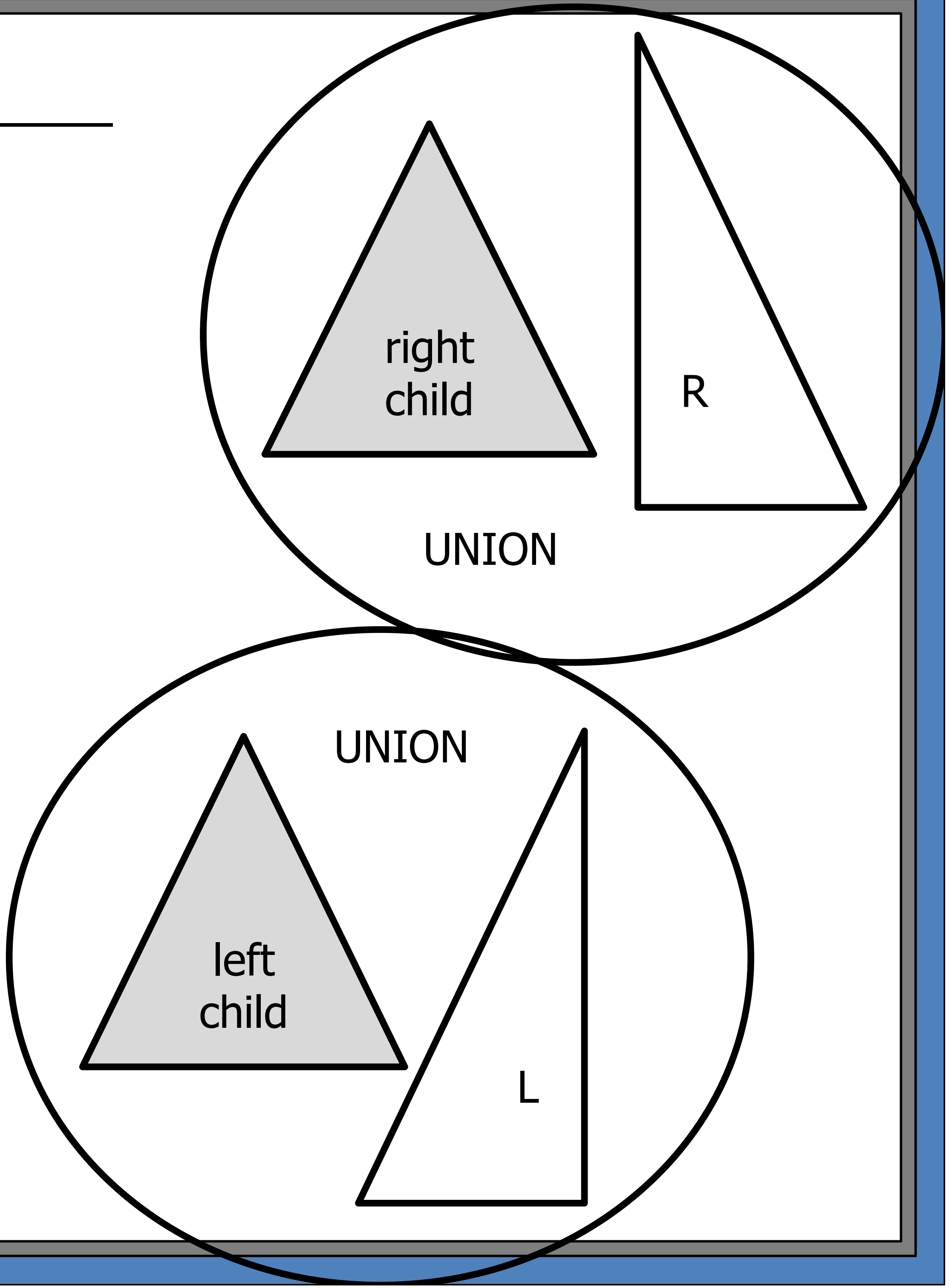
(L, G, x) = split(T2, key)

fork:

1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync

…



Parallel Sets

Union(T1, T2)

if T1 = null: **return** T2

if T2 = null: **return** T1

key = root(T1)

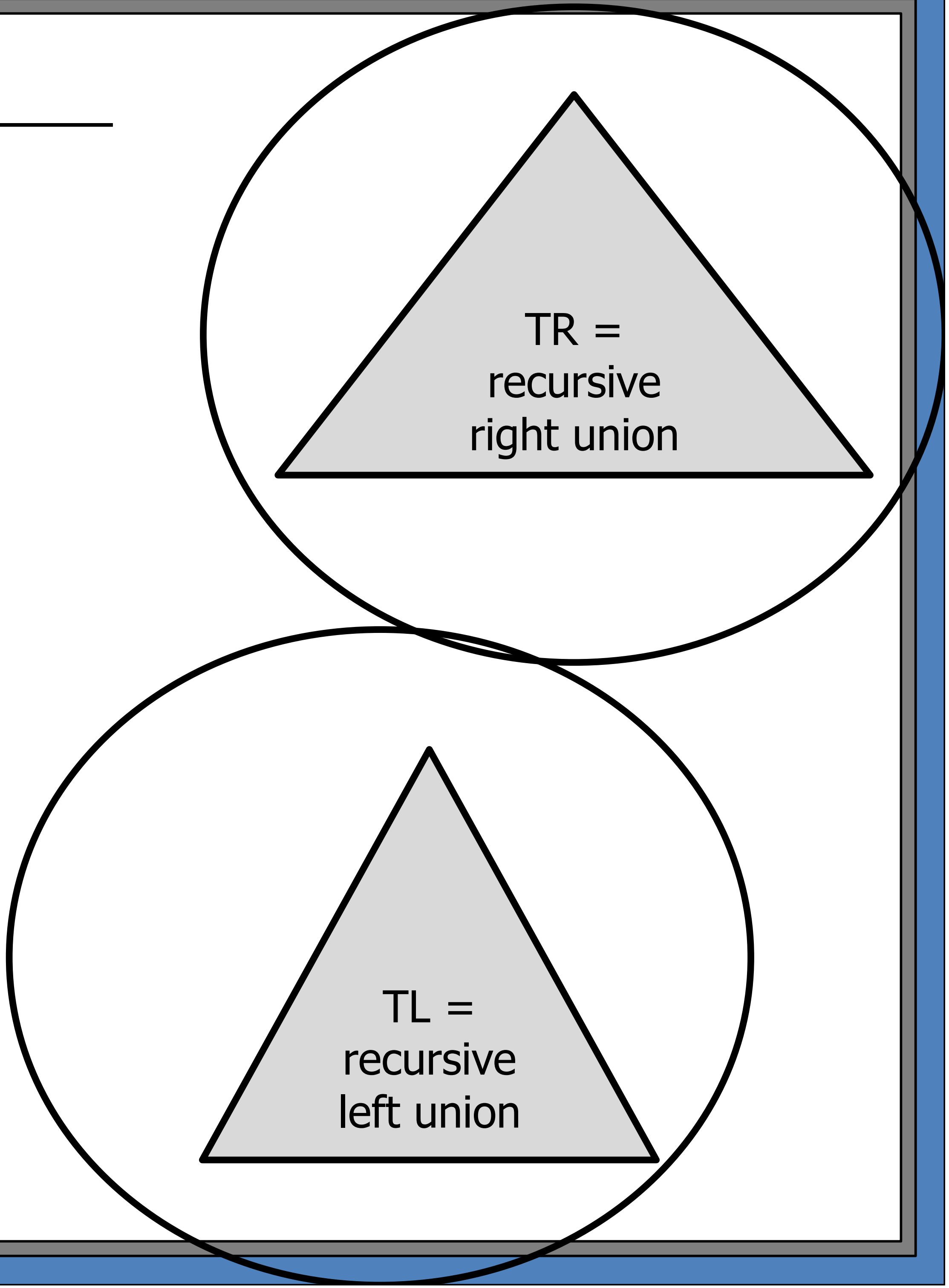
(L, G, x) = split(T2, key)

fork:

1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync

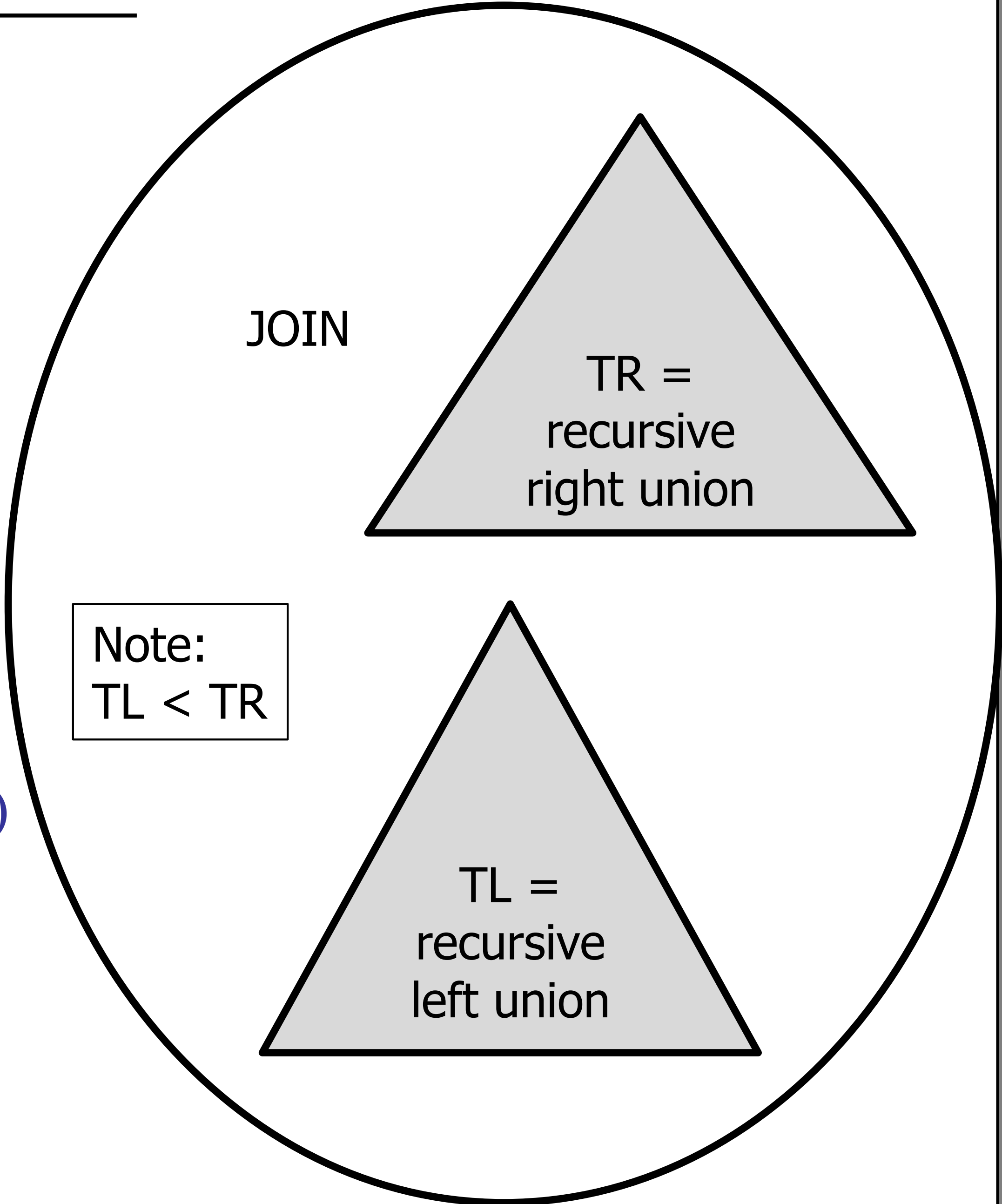
…



Parallel Sets

Union(T1, T2)

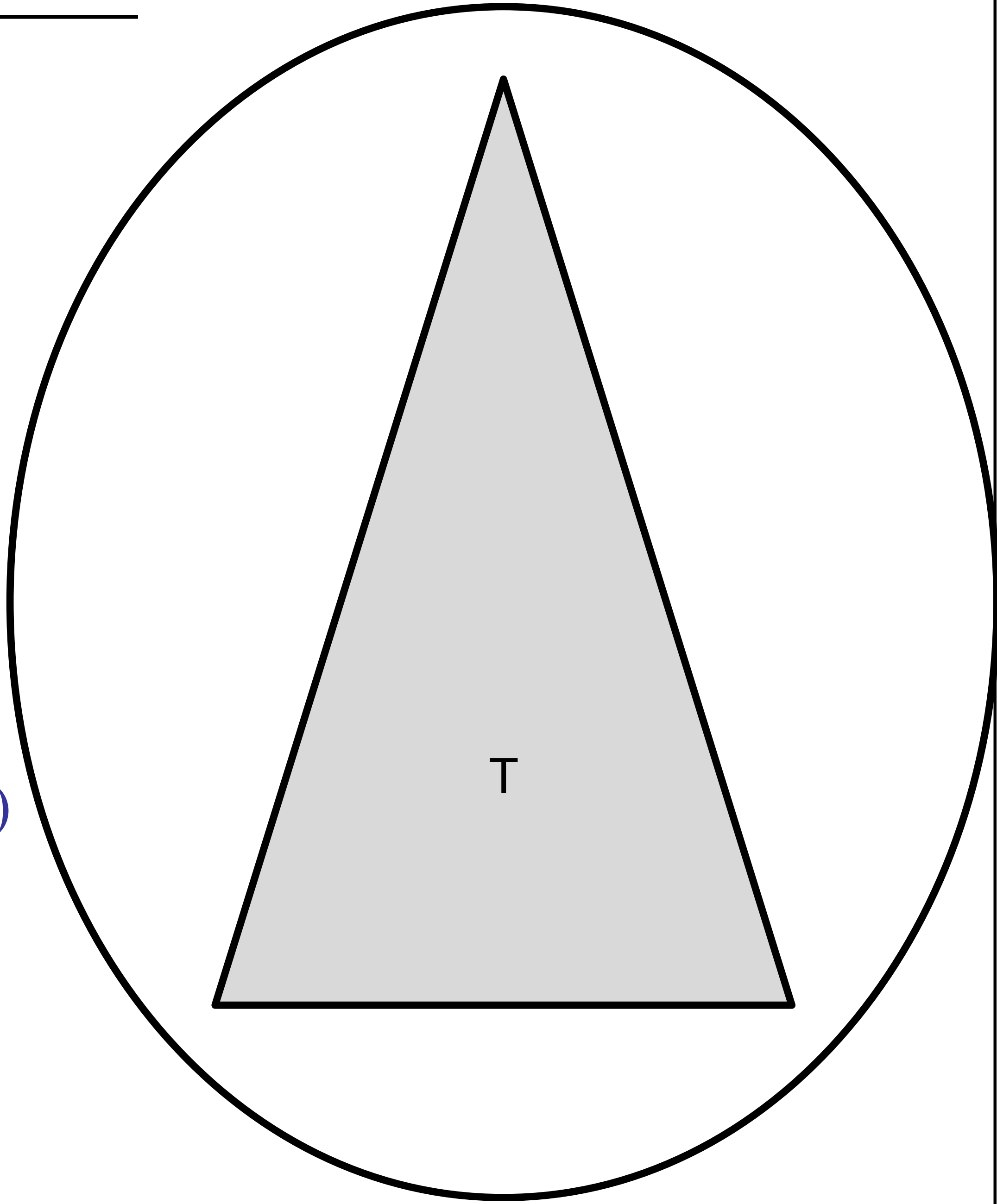
```
if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1.    TL = Union(key.left, L)
2.    TR = Union(key.right, R)
sync
T = join(TL, TR)
insert(T, key)
return T
```



Parallel Sets

Union(T1, T2)

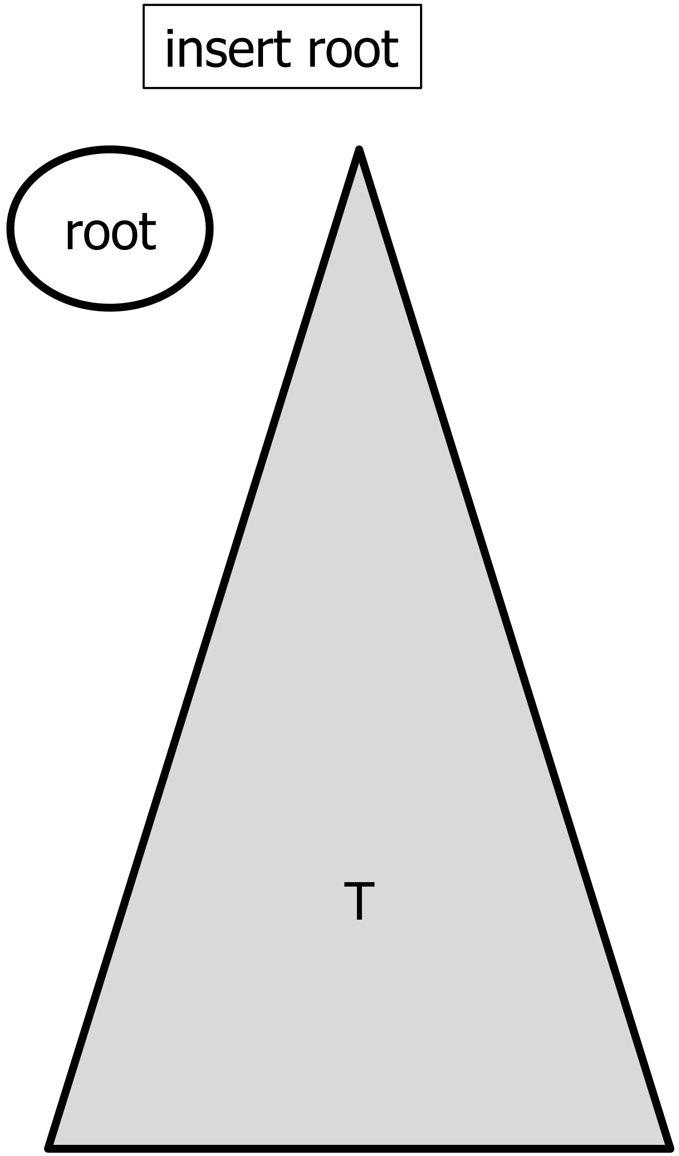
```
if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1.   TL = Union(key.left, L)
2.   TR = Union(key.right, R)
sync
T = join(TL, TR)
insert(T, key)
return T
```



Parallel Sets

Union(T1, T2)

```
if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1.   TL = Union(key.left, L)
2.   TR = Union(key.right, R)
sync
T = join(TL, TR)
insert(T, key)
return T
```



Work Analysis

Union(T1, T2)

```
if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1.  TL = Union(key.left, L)
2.  TR = Union(key.right, R)
sync
T = join(TL, TR)
insert(T, key)
return T
```

$O(1)$

Work Analysis

Union(T1, T2)

```
if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1.  TL = Union(key.left, L)
2.  TR = Union(key.right, R)
sync
T = join(TL, TR)
insert(T, key)
return T
```

$O(\log n + \log m)$

Work Analysis

Union(T1, T2)

```
if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1.  TL = Union(key.left, L)
2.  TR = Union(key.right, R)
sync
T = join(TL, TR)
insert(T, key)
return T
```

Recursive calls where T1 is half the size.

Work Analysis

Union(T1, T2)

if T1 = null: **return** T2

if T2 = null: **return** T1

key = root(T1)

(L, G, x) = split(T2, key)

fork:

1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync

T = join(TL, TR)

insert(T, key)

return T

Recursive calls where T1 is half the size.

$$\boxed{\begin{aligned} T(n,m) &= 2T(n/2,m) + O(\log n + \log m) \\ &= O(n \log m) \end{aligned}}$$

Work Analysis

Union(T1, T2)

if T1 = null: **return** T2

if T2 = null: **return** T1

key = root(T1)

(L, G, x) = split(T2, key)

fork:

1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync

T = join(TL, TR)

insert(T, key)

return T

Lying (a little):
Left and right subtrees are not exactly sized n/2.

Still true…

$$\boxed{\begin{aligned} T(n,m) &= 2T(n/2,m) + O(\log n + \log m) \\ &= O(n \log m) \end{aligned}}$$

Work Analysis

Union(T1, T2)

if T1 = null: **return** T2

if T2 = null: **return** T1

key = root(T1)

(L, G, x) = split(T2, key)

fork:

1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync

T = join(TL, TR)

insert(T, key)

return T

Be more careful

if m < n then:

Work = O(m log(n/m))

$$\begin{aligned} T(n,m) &= 2T(n/2,m) + O(\log n + \log m) \\ &= O(n \log m) \end{aligned}$$

Span Analysis

Union(T1, T2)

if T1 = null: **return** T2

if T2 = null: **return** T1

key = root(T1)

(L, G, x) = split(T2, key)

fork:

1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

Recursive calls where T1 is half the size.

sync

T = join(TL, TR)

insert(T, key)

return T

$$\boxed{\begin{aligned} S(n,m) &= T(n/2,m) + O(\log n + \log m) \\ &= O(\log^2 n) \end{aligned}}$$

Span Analysis

Union(T1, T2)

if T1 = null: **return** T2

if T2 = null: **return** T1

key = root(T1)

(L, G, x) = split(T2, key)

fork:

1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync

T = join(TL, TR)

insert(T, key)

return T

Use a different type of model / scheduler:

if m < n then:

Span = O(log n)

$$\begin{aligned} S(n,m) &= T(n/2,m) + O(\log n + \log m) \\ &= O(\log^2 n) \end{aligned}$$

Span Analysis

Union(T1, T2)

if T1 = null: **return** T2

if T2 = null: **return** T1

key = root(T1)

(L, G, x) = split(T2, key)

fork:

1. TL = Union(key.left, L)
2. TR = Union(key.right, R)

sync

T = join(TL, TR)

insert(T, key)

return T

Not in CS5234

Use a different type of model / scheduler:

if m < n then:

Span = O(log n)

$$\begin{aligned} S(n,m) &= T(n/2,m) + O(\log n + \log m) \\ &= O(\log^2 n) \end{aligned}$$

Other operations?

Other operations?

Intersection(T1, T2)

```
if T1 = null: return null
if T2 = null: return null
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1.   TL = Intersection(key.left, L)
2.   TR = Intersection(key.right, R)
sync
T = join(TL, TR)
if (x = key) then insert(T, key)
return T
```

Other operations?

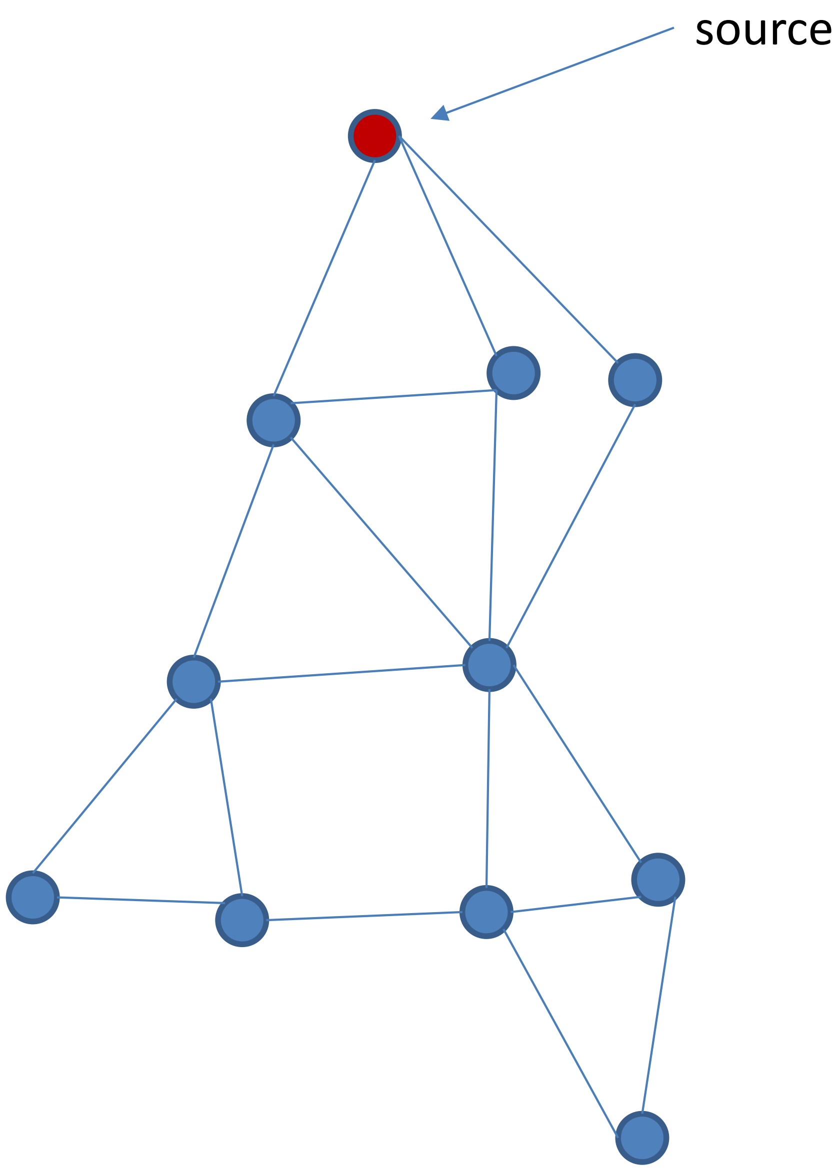
SetDifference(T1, T2)

```
if T1 = null: return T2
if T2 = null: return T1
key = root(T1)
(L, G, x) = split(T2, key)
fork:
1.   TL = Intersection(key.left, L)
2.   TR = Intersection(key.right, R)
sync
T = join(TL, TR)
if (x = null) then insert(T, key)
return T
```

Problem: Breadth First Search

Searching a graph:

- undirected graph G = (V,E)
- source node s

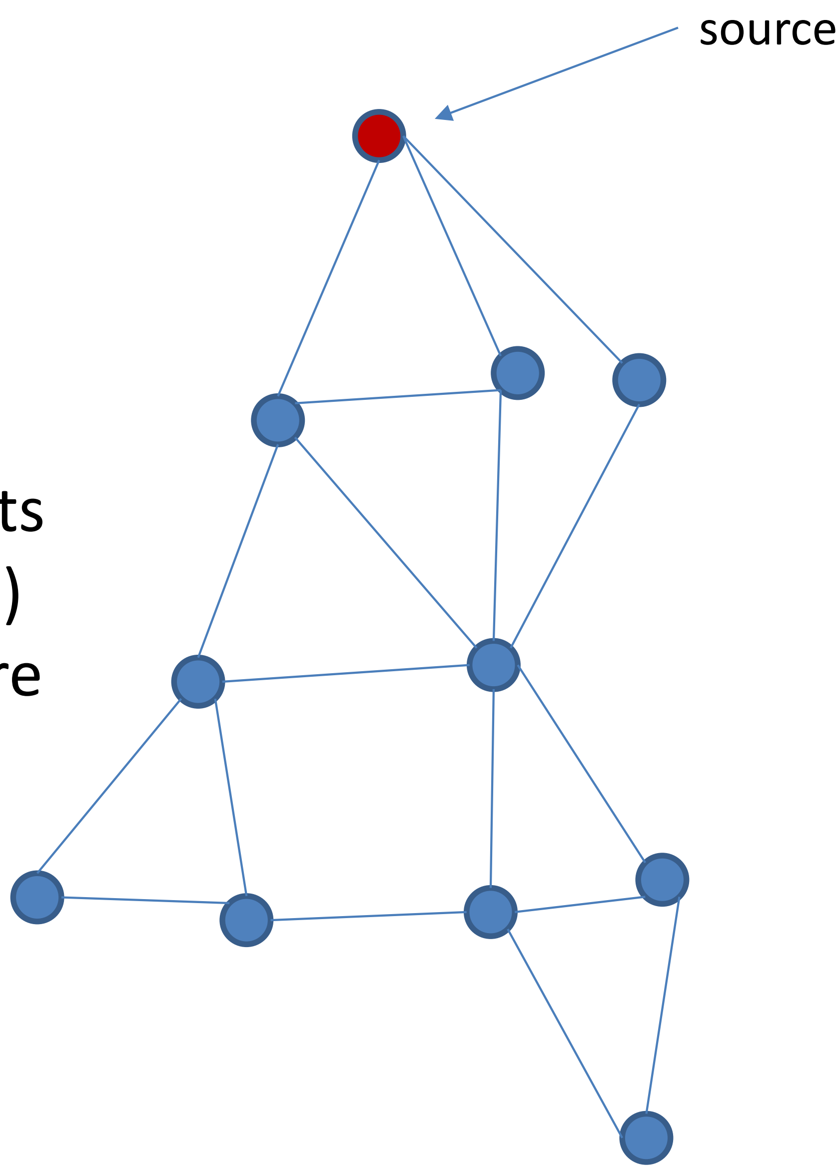


Problem: Breadth First Search

Searching a graph:

- undirected graph G = (V,E)
- source node s

- assume each node stores its adjacency list as a (parallel) set, using the data structure from before.

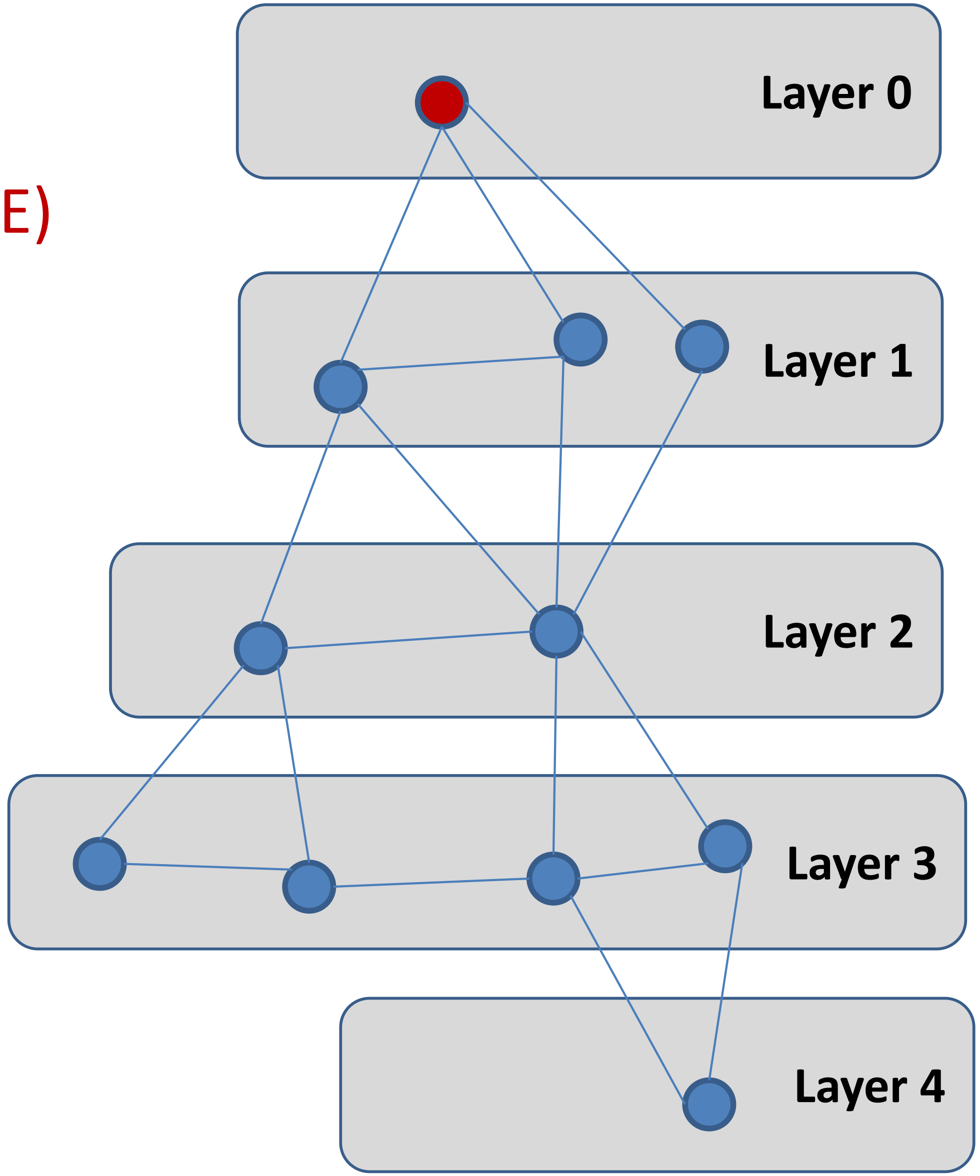


Problem: Breadth First Search

Searching a graph:

- undirected graph G = (V,E)
- source node s

Layer-by-layer…



Sequential Algorithm

```
BFS(G, s)
    F = {s}
    repeat until F = {}
        F' = {}
        for each u in F:
            visited[u] = true
            for each neighbor v of u:
                if (visited[v] = false) then F'.insert(v)
        F = F'
```

Sequential Algorithm

```
BFS(G, s)
    F = {s}
    repeat until F = {}
        F' = {}
        for each u in F:
            visited[u] = true
            for each neig
                if (vis
        F = F'
```

Problems to solve:

- need to do parallel exploration of the frontier
- visited is hard to maintain in parallel

Parallel Algorithm

parBFS(G, s)

F = {s}

D = {}

repeat until F = {}

D = Union(D, F)

F = ProcessFrontier(F)

F = SetSubtraction(F, D)

F and D are parallel sets, built using the parallel data structure we saw earlier!

Parallel Algorithm

```
parBFS(G, s)
    F = {s}
    D = {}
    repeat until F = {}
        D = Union(D, F)
        F = ProcessFrontier(F)
        F = SetSubtraction(F, D)
```

Mark everything already explored as done.

Parallel Algorithm

```
parBFS(G, s)
    F = {s}
    D = {}
    repeat until F = {}
        D = Union(D, F)
        F = ProcessFrontier(F)
        F = SetSubtraction(F, D)
```

Mark everything already explored as done.

Explore all the neighbors of every node in F.

Parallel Algorithm

parBFS(G, s)

F = {s}

D = {}

repeat until F = {}

D = Union(D, F)

F = ProcessFrontier(F)

F = SetSubtraction(F, D)

Mark everything already explored as done.

Explore all the neighbors of every node in F.

Remove already visited nodes from the new frontier.

Parallel Algorithm

```
ProcessFrontier(F)
    if |F| = 1 then
        u = root(F)
        return u.neighbors
    else
        (F1, F2) = divide(F)
        fork:
         1.  F1 = ProcessFrontier(F1)
         2.  F2 = ProcessFrontier(F2)
         sync
         return Union(F1, F2)
```

Base case: return the set containing the neighbors of one node.

Parallel Algorithm

```
ProcessFrontier(F)
    if |F| = 1 then
        u = root(F)
        return u.neighbors
    else
        (F1, F2) = divide(F)
        fork:
         1.  F1 = ProcessFrontier(F1)
         2.  F2 = ProcessFrontier(F2)
         sync
         return Union(F1, F2)
```

Base case: return the set containing the neighbors of one node.

Divide the set (approximately) in half.

Parallel Algorithm

```
ProcessFrontier(F)
    if |F| = 1 then
        u = root(F)
        return u.neighbors
    else
        (F1, F2) = divide(F)
        fork:
          1.  F1 = ProcessFrontier(F1)
          2.  F2 = ProcessFrontier(F2)
        sync
        return Union(F1, F2)
```

Base case: return the set containing the neighbors of one node.

Divide the set (approximately) in half.

Recursively process the two frontiers.

Parallel Algorithm

```
ProcessFrontier(F)
    if |F| = 1 then
        u = root(F)
        return u.neighbors
    else
        (F1, F2) = divide(F)
        fork:
         1.  F1 = ProcessFrontier(F1)
         2.  F2 = ProcessFrontier(F2)
         sync
         return Union(F1, F2)
```

Base case: return the set containing the neighbors of one node.

Divide the set (approximately) in half.

Recursively process the two frontiers.

Merge the two frontiers and return.

Work Analysis

n = nodes in F
m = # adjacent edges to F

```
ProcessFrontier(F)
    if |F| = 1 then
        u = root(F)
        return u.neighbors
    else
        (F1, F2) = divide(F)
        fork:
         1.  F1 = ProcessFrontier(F1)
         2.  F2 = ProcessFrontier(F2)
         sync
         return Union(F1, F2)
```

Work Analysis

n = nodes in F
m = # adjacent edges to F

```
ProcessFrontier(F)
    if |F| = 1 then
        u = root(F)
        return u.neighbors
    else
        (F1, F2) = divide(F)
        fork:
         1.  F1 = ProcessFrontier(F1)
         2.  F2 = ProcessFrontier(F2)
         sync
         return Union(F1, F2)
```

$O(1)$

Work Analysis

n = nodes in F
m = # adjacent edges to F

```
ProcessFrontier(F)
    if |F| = 1 then
        u = root(F)
        return u.neighbors
    else
        (F1, F2) = divide(F)
        fork:
         1.  F1 = ProcessFrontier(F1)
         2.  F2 = ProcessFrontier(F2)
         sync
         return Union(F1, F2)
```

$O(1)$

$O(\log n)$

Work Analysis

n = nodes in F
m = # adjacent edges to F

```
ProcessFrontier(F)
    if |F| = 1 then
        u = root(F)
        return u.neighbors
    else
        (F1, F2) = divide(F)
        fork:
         1.  F1 = ProcessFrontier(F1)
         2.  F2 = ProcessFrontier(F2)
         sync
         return Union(F1, F2)
```

$O(1)$

$O(\log n)$

Two recursive calls
of size approximately n/2.

Work Analysis

n = nodes in F
m = # adjacent edges to F

ProcessFrontier(F)

```
if |F| = 1 then
    u = root(F)                          ← O(1)
    return u.neighbors
else
    (F1, F2) = divide(F)                 ← O(log n)
    fork:
     1.  F1 = ProcessFrontier(F1)        ← Two recursive calls
     2.  F2 = ProcessFrontier(F2)          of size approximately n/2.
    sync
    return Union(F1, F2)                 ← O(m log m)
```

$O(1)$

$O(\log n)$

Two recursive calls of size approximately n/2.

$O(m \log m)$

$$
\begin{aligned}
W(n,m) &= 2W(n/2,m) + O(m\log m) + O(\log n) \\
&= O(m \log n \log m) \\
&= O(m \log^2 n)
\end{aligned}
$$

u = root(F) $O(1)$

return u.neighbors

else

(F1, F2) = divide(F) $O(\log n)$

fork:

1. F1 = ProcessFrontier(F1)
2. F2 = ProcessFrontier(F2)

Two recursive calls of size approximately n/2.

sync

return Union(F1, F2) $O(m \log m)$

Span Analysis

n = nodes in F
m = # adjacent edges to F

```
ProcessFrontier(F)
    if |F| = 1 then
        u = root(F)
        return u.neighbors
    else
        (F1, F2) = divide(F)
        fork:
         1.   F1 = ProcessFrontier(F1)
         2.   F2 = ProcessFrontier(F2)
         sync
         return Union(F1, F2)
```

$O(1)$

$O(\log n)$

One recursive calls
of size approximately n/2.

$O(\log^2 m)$

$$
\begin{aligned}
S(n,m) &= S(n/2,m) + O(\log^2 m) + O(\log n) \\
&= O(\log n \log^2 m) \\
&= O(\log^3 n)
\end{aligned}
$$

```
    u = root(F)
    return u.neighbors
else
    (F1, F2) = divide(F)
    fork:
     1.  F1 = ProcessFrontier(F1)
     2.  F2 = ProcessFrontier(F2)
    sync
    return Union(F1, F2)
```

$O(1)$

$O(\log n)$

One recursive calls
of size approximately n/2.

$O(\log^2 m)$

Parallel Algorithm

parBFS(G, s)

F = {s}

D = {}

repeat until F = {}

D = Union(D, F)

F = ProcessFrontier(F)

F = SetSubtraction(F, D)

Work: $O(m \log^2 n)$

Span: $O(\log^3 n)$

Work Analysis

parBFS(G, s)

F = {s}

D = {}

repeat until F = {}

D = Union(D, F) $\leftarrow O(m \log n)$

F = ProcessFrontier(F) $\leftarrow O(m \log^2 n)$

F = SetSubtraction(F, D) $\leftarrow O(m \log n)$

Note: every edge appears in at most two iterations!

Note: every node appears in at most one frontier.

F_j = number of nodes in frontier in jth iteration.

Work Analysis

$$T_1(n, m) = O(m \log^2 n)$$

parBFS(G, s)

F = {s}

D = {}

repeat until F = {}

D = Union(D, F) ← $O(m \log n)$

F = ProcessFrontier(F) ← $O(m \log^2 n)$

F = SetSubtraction(F, D) ← $O(m \log n)$

Note: every edge appears in at most two iterations!

Note: every node appears in at most one frontier.

F_j = number of nodes in frontier in jth iteration.

Span Analysis

parBFS(G, s)

F = {s}

D = {}

repeat until F = {}

D = Union(D, F) $O(\log^2 m)$

F = ProcessFrontier(F) $O(\log^3 m)$

F = SetSubtraction(F, D) $O(\log^2 m)$

Assume the graph has diameter D.

$$T_\infty = D \log^3 m$$

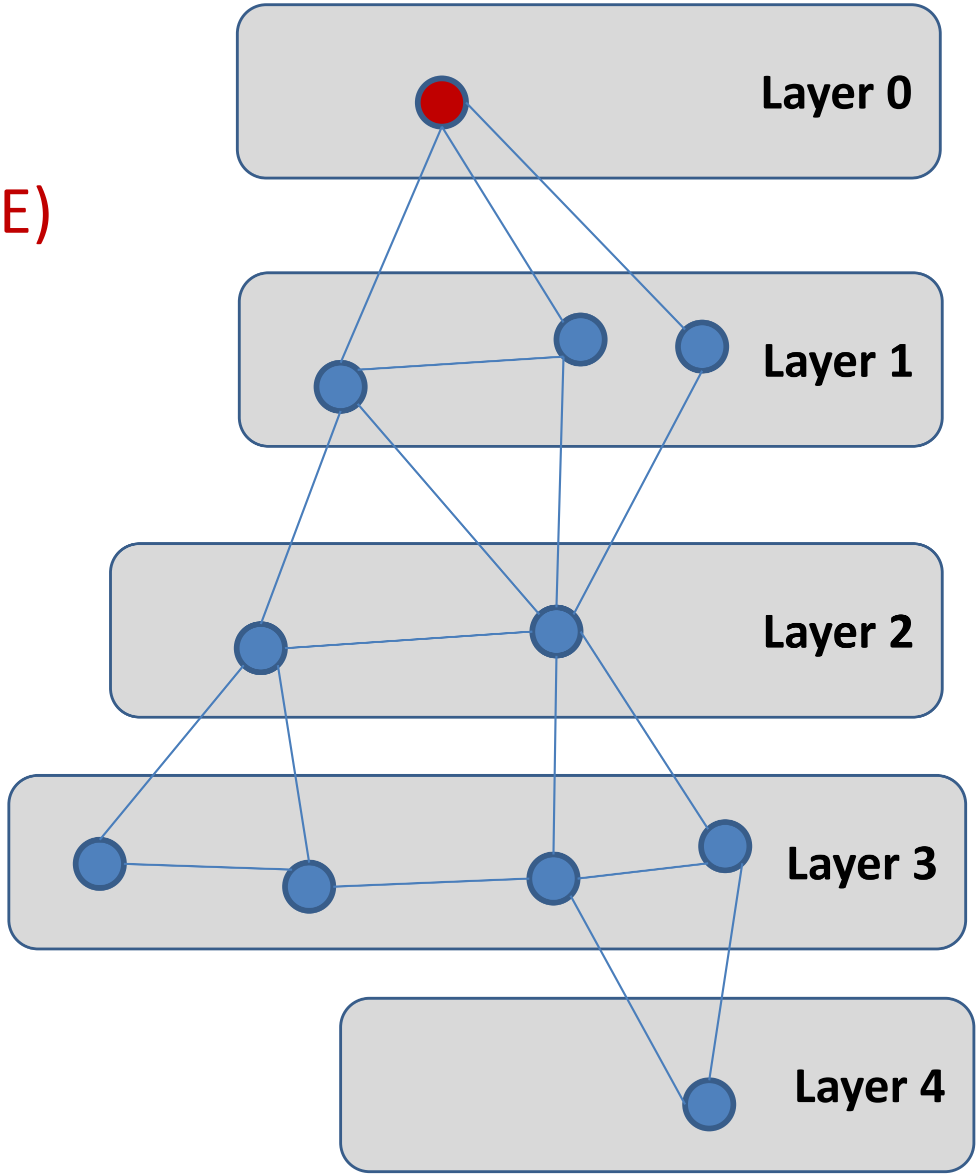
Hard to do better than D.

Problem: Breadth First Search

Searching a graph:

- undirected graph G = (V,E)
- source node s

Layer-by-layer…

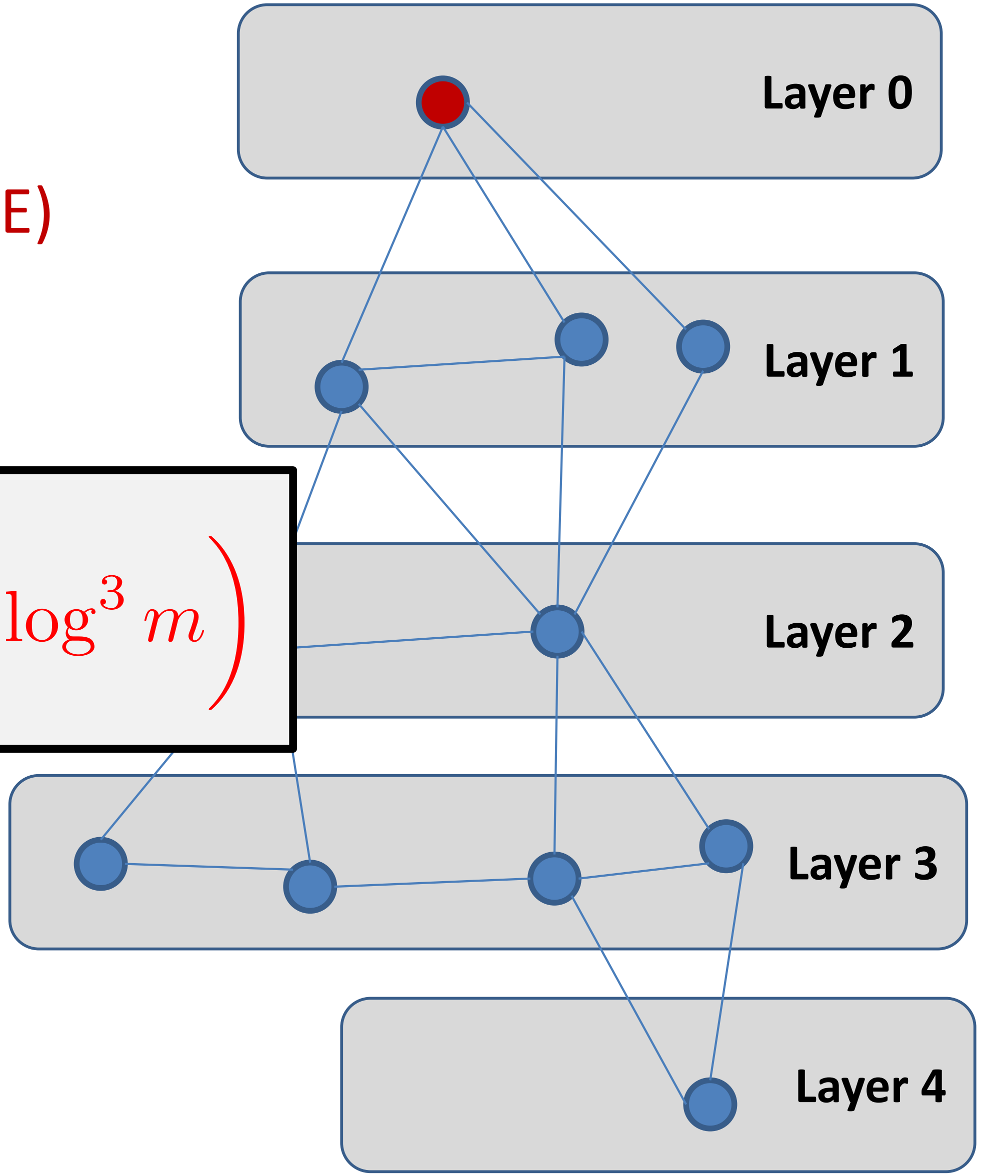


Problem: Breadth First Search

Searching a graph:

- undirected graph G = (V,E)
- source node s

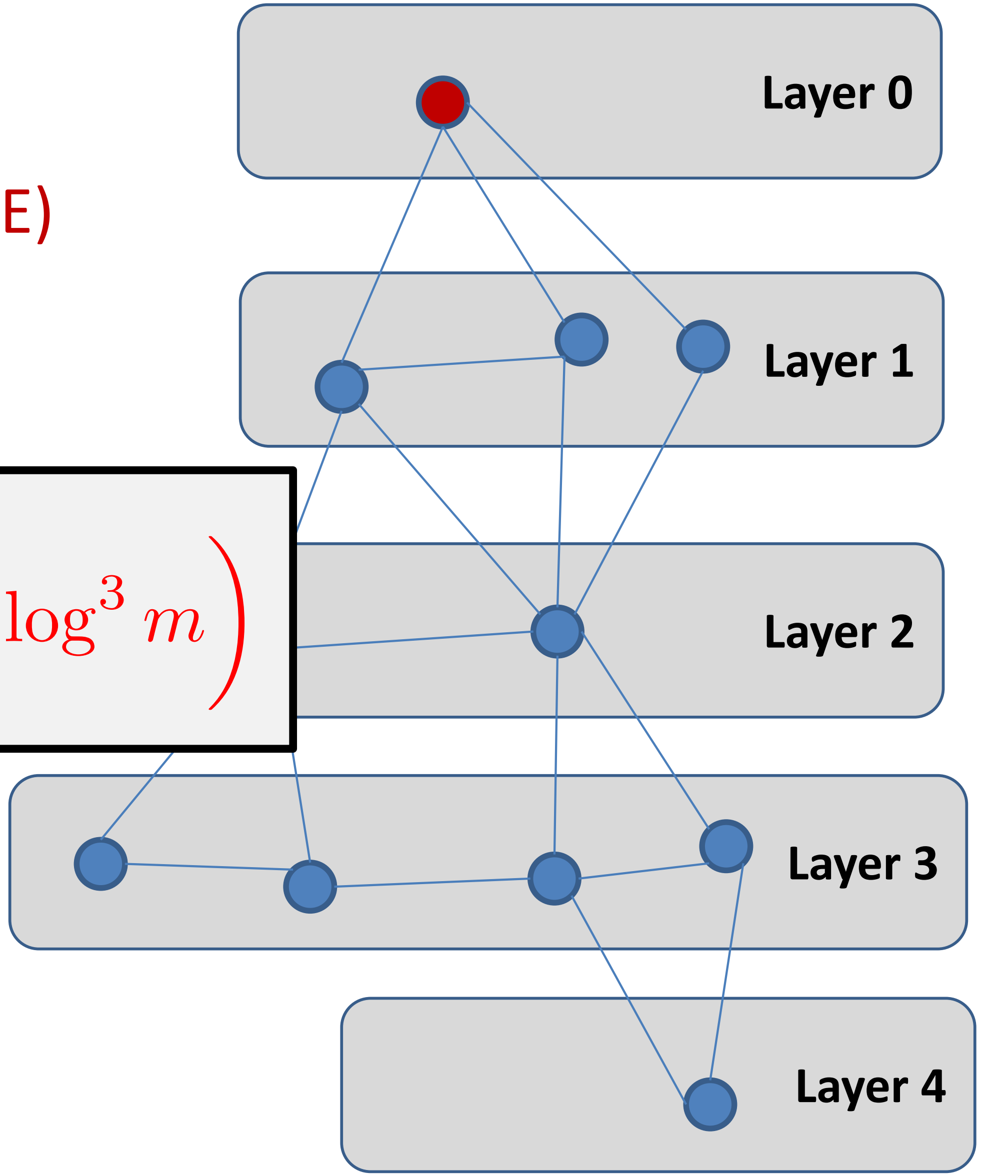
$$T_p = O\left(\frac{m \log^2 n}{p} + D \log^3 m\right)$$



Problem: Breadth First Search

Searching a graph:

- undirected graph G = (V,E)
- source node s



$$T_p = O\left(\frac{m \log^2 n}{p} + D \log^3 m\right)$$

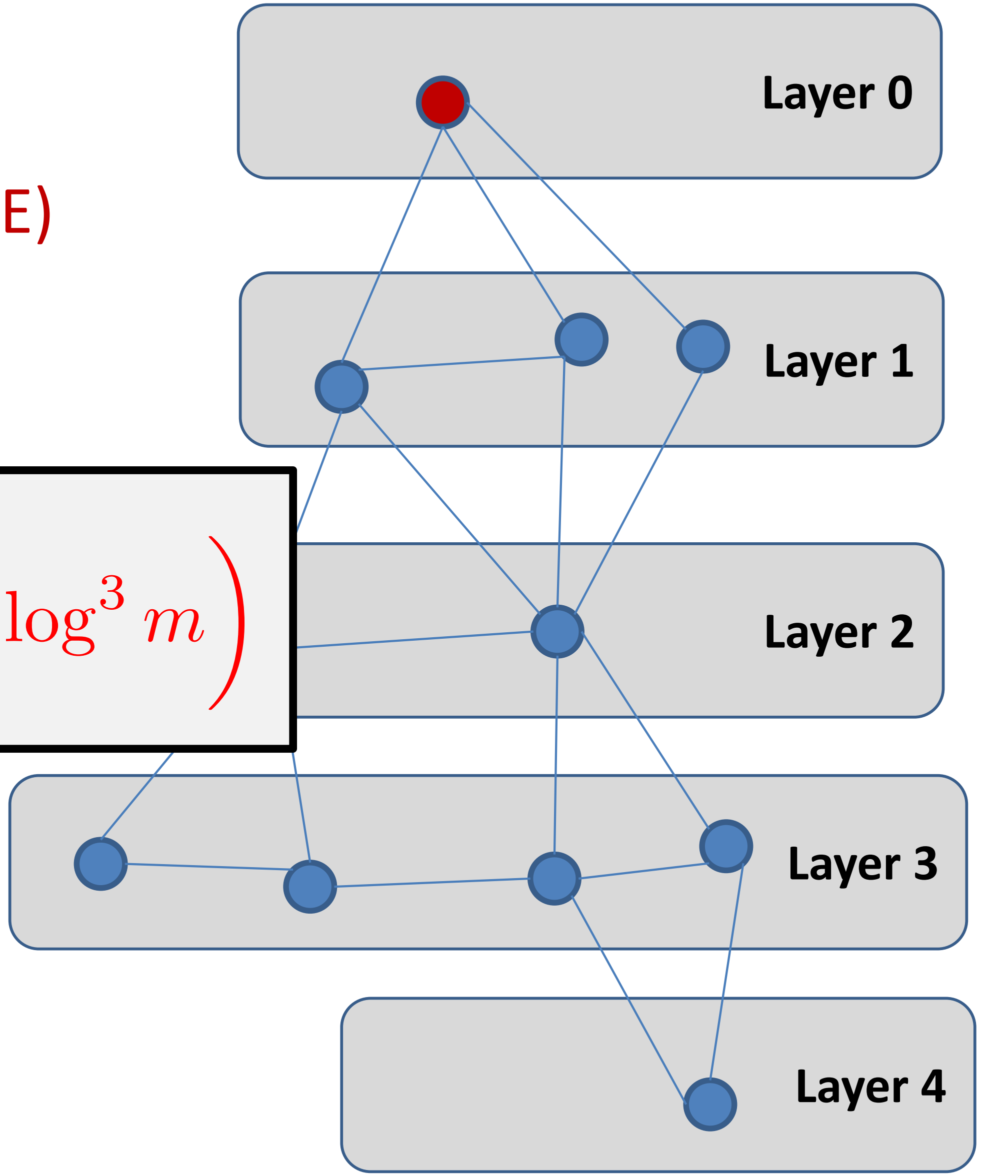
<u>Interpretation:</u>
With a *good* scheduler and enough processors, you can perform a BFS in time *roughly* proportional to the diameter.

Problem: Breadth First Search

Searching a graph:

- undirected graph G = (V,E)
- source node s

$$T_p = O\left(\frac{m \log^2 n}{p} + D \log^3 m\right)$$



Caveat:
Only useful if $p > \log^2 n$.

Problem: Depth First Search

Searching a graph:

- undirected graph $G = (V,E)$
- source node s
- search graph in depth-first order

➔ Best we know is $\Omega(n)$

Why does DFS seem so much harder than BFS?

Summary

Today: Parallelism

Models of Parallelism

- How to predict the performance of algorithms?

Some simple examples…

Sorting

- Parallel MergeSort

Trees and Graphs

Last Week: Caching

Breadth-First-Search

- *Sorting your graph*

MIS

- *Luby's Algorithm*
- *Cache-efficient implementation*

MST

- *Connectivity*
- *Minimum Spanning Tree*