CS2020
Data Structures and Algorithms

Last Day of CS2020!

Last Three Classes

Last Friday

– Geometry

More details when you take Computational Geometry

Last Wednesday

– Network Flows

More details in CS 4234 (Optimization Algorithms) and CS5234.

Today

– Parallel Algorithms

More details when you take a Parallel Computing module

Announcements

Clickers

- Return at the end of lecture today.
- Don’t forget!
- If you have lost your clicker:

  S$105 / clicker

  Come talk to us ASAP…

Announcements

Anonymous Feedback

1. For SoC: NUS Student Feedback System
    - Due April 22nd
    - Tell the departmental administrators what you think of CS2020.

2. For me: Coursemology Survey
    - How can I improve CS2020 next year?
    - What should we do differently?

Feedback for me

Question 1: **General**

1. Feedback on me
2. Feedback on your tutor

Question 2: **Organization**

1. How did DGs work for you?
2. What did you think of problem sessions?

Question 3: **Technology**

1. What did you think of NB?
2. What did you think of Coursemology?
3. What did you think of the clickers?

Many thanks to our great team of tutors

Many thanks to our great team of tutors

weird
blue
pixels?

Final Exam

Final Exam

Wednesday April 25: 5pm

- Location: Multi-Purpose Sports Hall 1-B
- Same rules as Quiz 1 and Quiz 2:
  - Closed book
  - Two double-sided pieces of paper
- Three-way focus:
  - Basic algorithms and data structures
  - Java
  - Problem solving
- All the material from CS2020

Breakdown by topics

	Algorithms / Theory	Java	Problem solving
Quiz 1	45	35	20
Coding Quiz	20	80	0
Quiz 2	20	0	80
Total:	***28***	***38***	***33***

	Algorithms / Theory	Java	Problem solving
Final Exam	35	25	40

Exam Topics

Theory:

- Asymptotic analysis
- Simple recurrences
- Simple probability

Exam Topics

Algorithms and data structures, part 1:

- Abstract Data Types
  - Bags, Lists, Stacks, Queues
- Divide-and-conquer
  - Binary search
  - Peak finding
- Sorting
  - BubbleSort, InsertionSort, SelectionSort, MergeSort, QuickSort
  - Reversal Sorting, The Birthday Party, etc.

Exam Topics

Algorithms and data structures, part 2:

- Trees
  - Binary search trees
  - Tries
  - AVL trees
  - Augmented Trees
    - Order statistics trees (Select, Rank)
    - Interval trees
    - Range trees
  - Skip lists
  - Heaps
  - Union-Find

Exam Topics

Algorithms and data structures, part 3:

- Hashing
  - Direct access tables
  - Chaining
  - Open addressing
  - Table resizing
  - Basic hash functions
  - Simple probability
  - Simple amortized analysis
  - Bloom filters

Exam Topics

Algorithms and data structures, part 4:

– Basic graphs
  - Formats: adjacency list/matrix
  - BFS, DFS, etc.
– Shortest paths
  - Dijkstra's Algorithm
  - Bellman-Ford
  - Special cases
– Minimum spanning trees
  - Cut Property, Cycle Property
  - Prim's, Kruskal's Algorithms

Exam Topics

Advanced topics:

- Dynamic programming
  - Simple problems
  - Floyd-Warshall All-Pairs Shortest Path
- Geometry
  - BSP, kd-tree, line intersection
- Flow networks
  - Ford-Fulkerson Algorithm
- Parallel Algorithms
  - Parallel MergeSort

Exam Topics

Java:

- Object-oriented programming
  - Basic principles and implementation in Java
- Basic Java
  - classes and inheritance: class, interface, implements, extends
  - access control: public, private, protected, static
  - simple error handling: exceptions
  - Generics
  - Comparable
  - Iterators
  - etc…..

Advice

Much like Quiz 1:

- Review how the algorithms works.
- Review the different techniques.
- Review Java details.

Think about common themes:

- For example: Binary search
- For example: Augmenting data structures
- For example: Developing proper graph models
- And others…

Advice

Think about trade-offs:

- When do you use Bellman-Ford vs. Dijkstra’s?
- When do you use Chaining vs. Open addressing?
- Etc…

Don’t stress…

- Be happy.

Exam review

Problem session today!

Many tutor’s holding special review sessions…

Final Exam

Wednesday April 25: 5pm

- Location: Multi-Purpose Sports Hall 1-B
- Same rules as Quiz 1 and Quiz 2:
  - Closed book
  - Two double-sided pieces of paper
- Three-way focus:
  - Basic algorithms and data structures
  - Java
  - Problem solving
- All the material from CS2020

What next?

What next?

- Algorithms modules:
  - CS3230: Design and Analysis of Algorithms
  - CS3210: Parallel Computing
  - CS3233: Competitive Programming
  - CS4231: Parallel and Distributed Algorithms
  - CS4234: Optimization
  - CS5234: Combinatorial and Graph Algorithms
  - CS5237: Computational Geometry
- Theory:
  - CS4232: Theory of Computation
  - CS5230: Computational Complexity

What next?

- Software engineering modules:
  - CS2103: Software engineering
  - CS4211: Formal methods for software engineering
  - CS4218: Software testing and debugging

- System design and programming modules:
  - CS3216: Software development on evolving platforms
  - CS3217: Software engineering on modern application platforms

What next?

- Specialized modules:
  - Distributed Systems
  - Computer Security
  - Game Design
  - Computer Graphics
  - Machine Learning
  - Computational Biology
  - Wireless computing and sensor networks
  - Etc…

Today

Parallel Algorithms

Goal:

- Develop a parallel sorting algorithm.
- Your challenge: parallelize the rest of the algorithms we have studied.

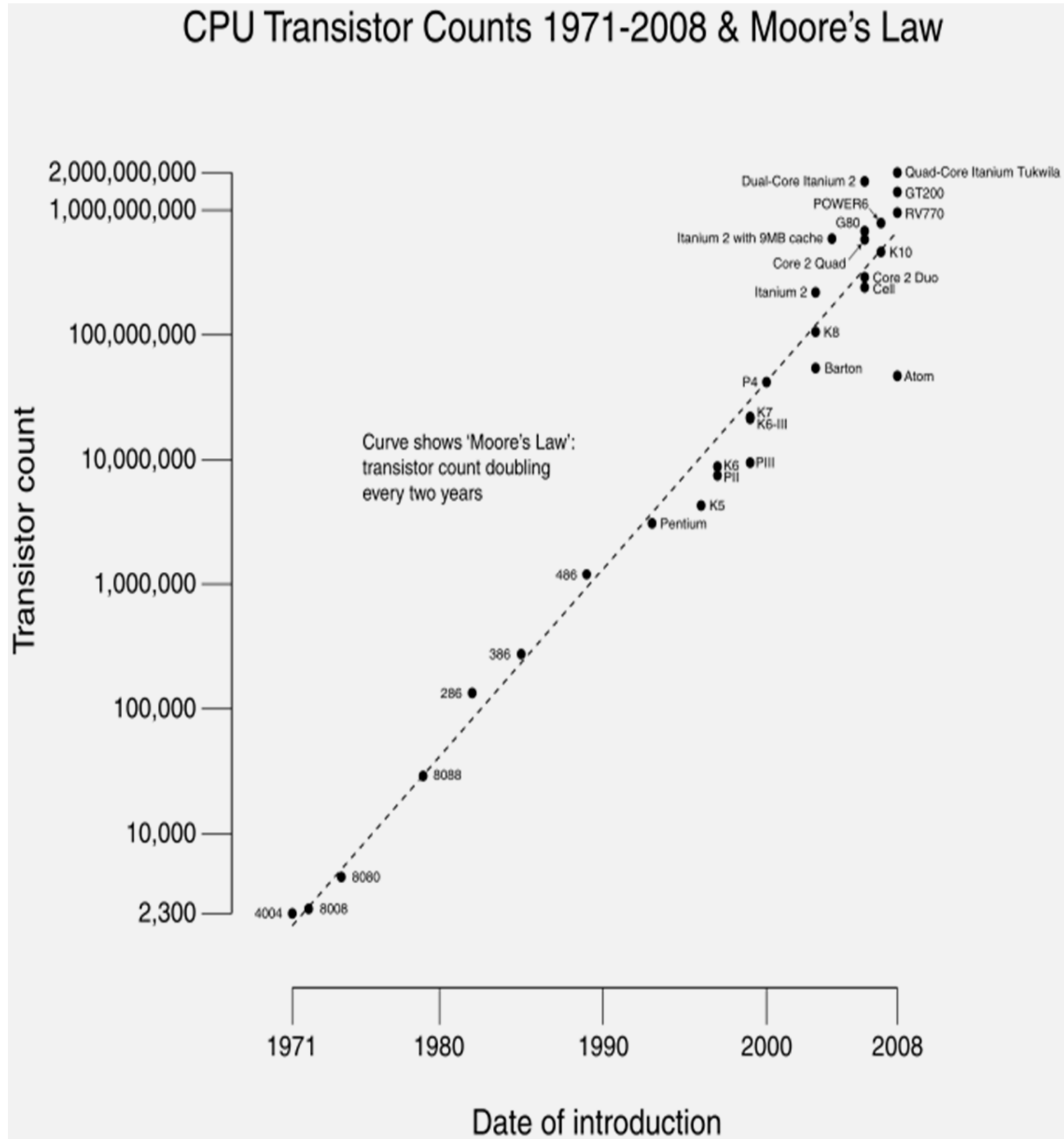
Parallel Algorithms

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

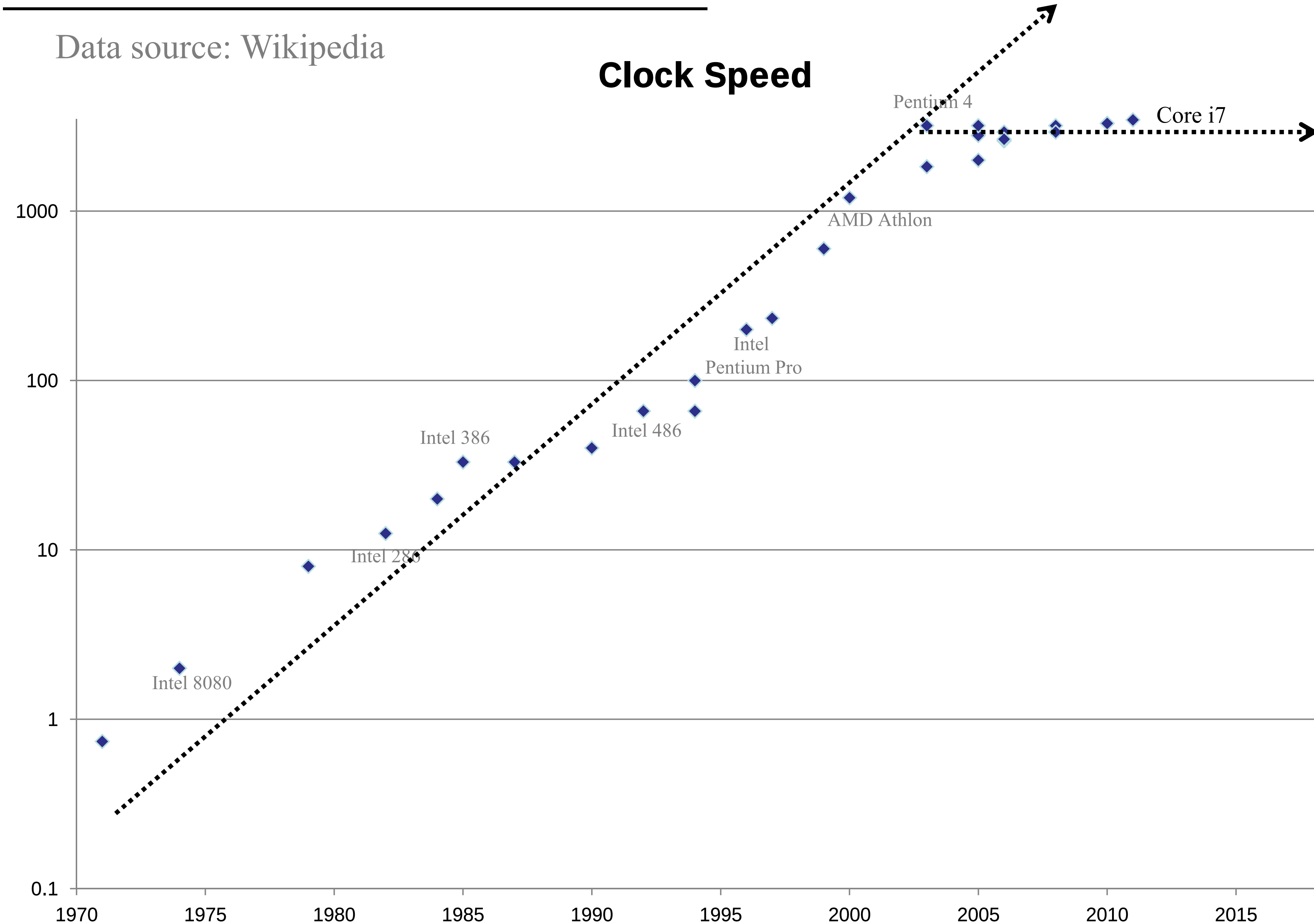
Clock speed?

- 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

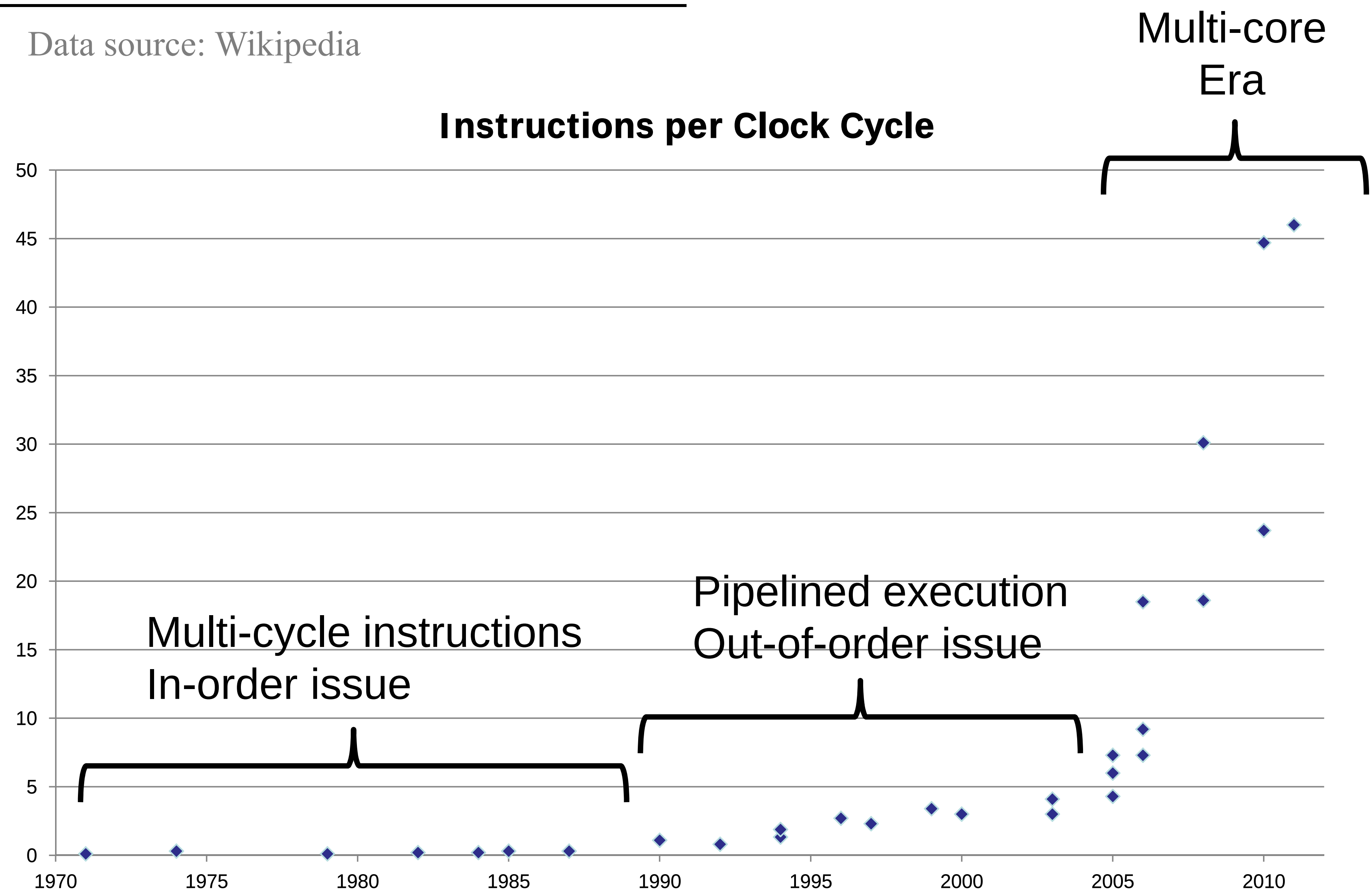
Conclusion:

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

What do we do?

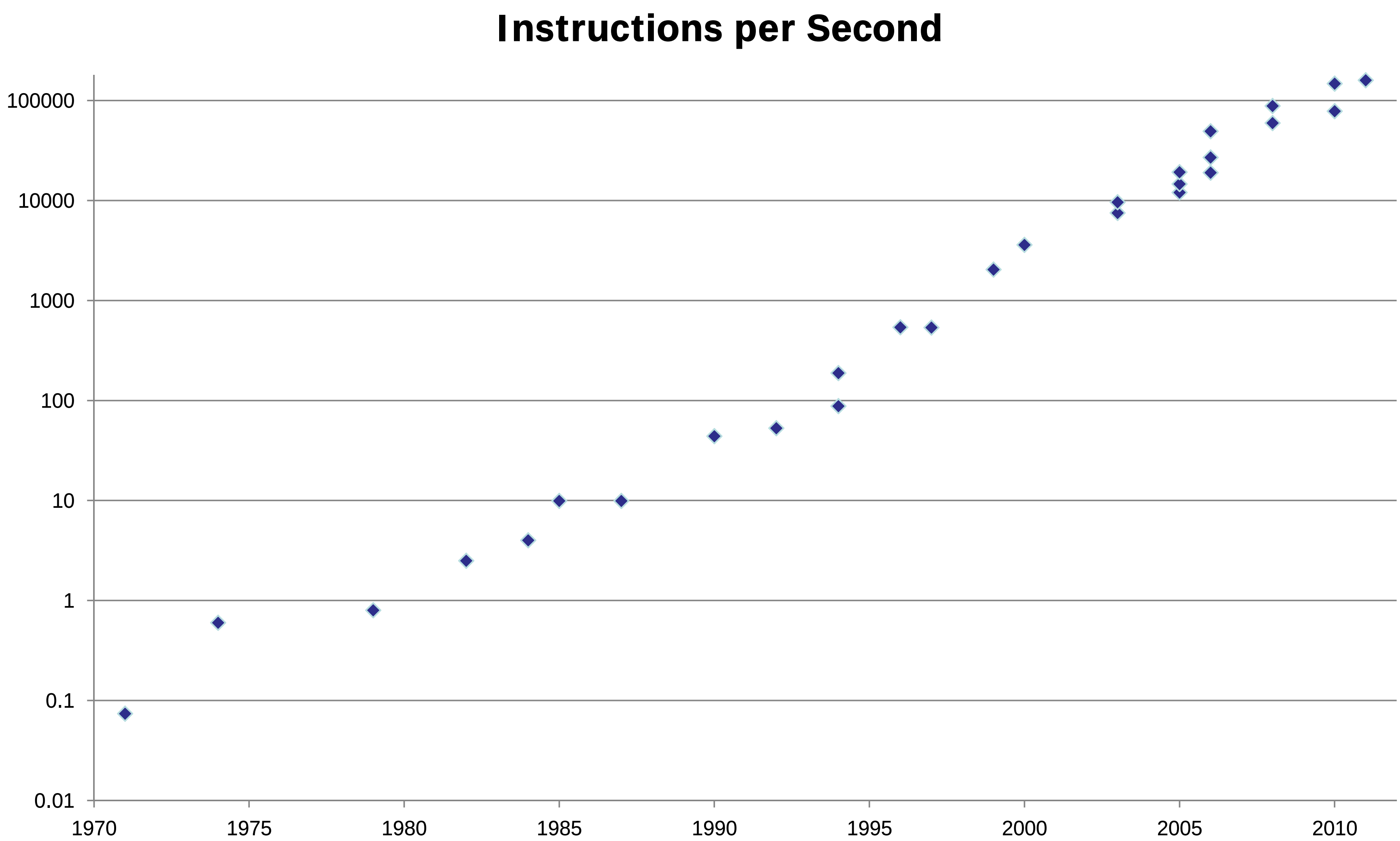
Parallel Algorithms

Data source: Wikipedia



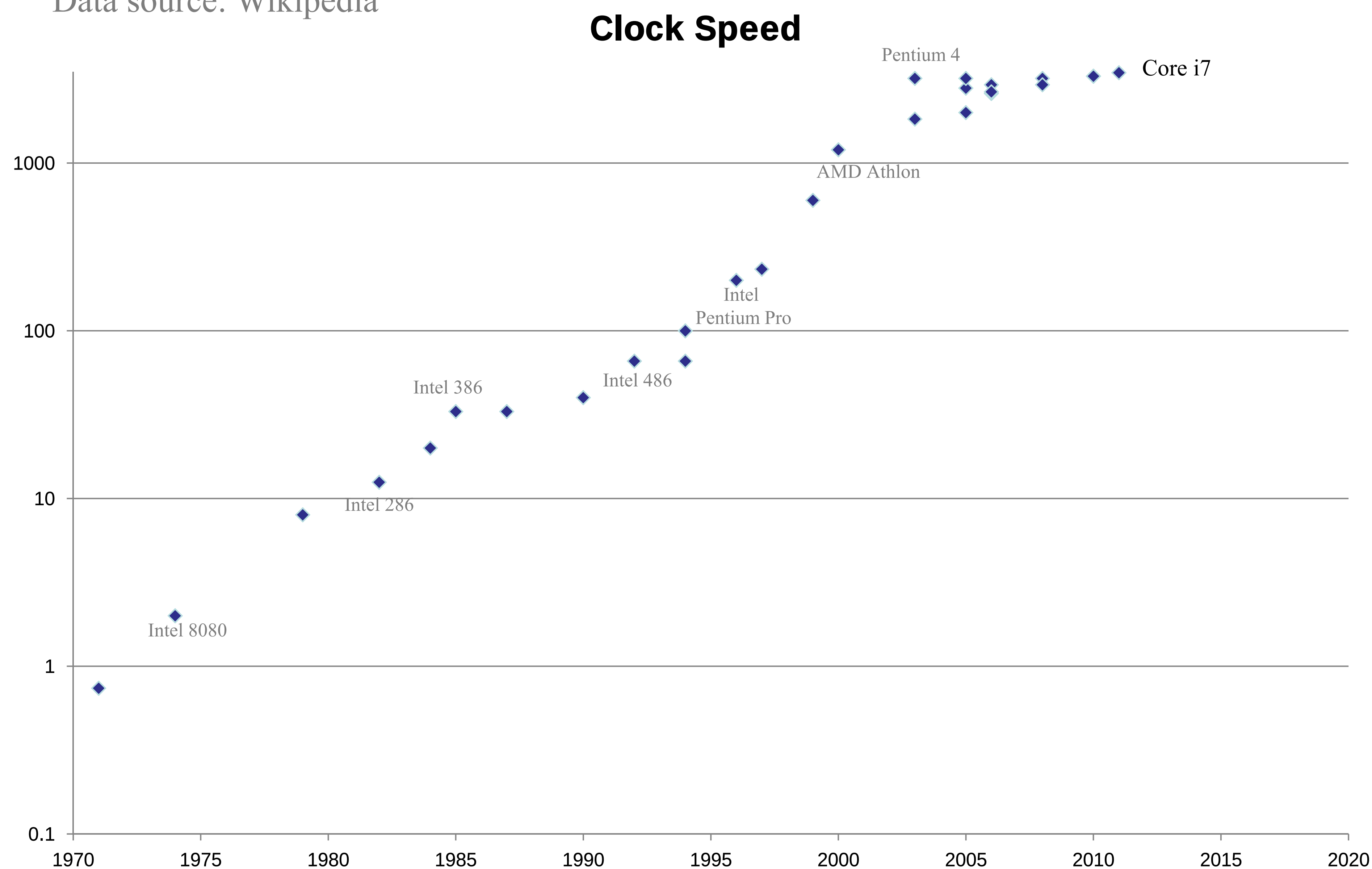
Parallel Algorithms

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

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.

Today

Parallel Algorithms

- Developing an algorithm with lots of parallelism.
- Analyzing parallelism.
- MergeSort

Not going to have time to talk about:

- Parallel programming
- Race conditions
- Locks
- Etc….

Questions to ask:

- What steps can we execute at the same time?
- How much parallelism is possible?
- How fast will it run:
  - On a single processor machine?
  - On an 8-core machine?
  - On a really, really big supercomputer?

Models

Sequential algorithms:

Easy: execute one step after another…

Models

Parallel algorithm models:

- PRAM (EREW/CREW/CRCW)
- SIMD/MIMD
- BSP
- Shared memory
- Message passing
- Dataflow
- LogP
- etc….

Models

Dynamic Multithreading

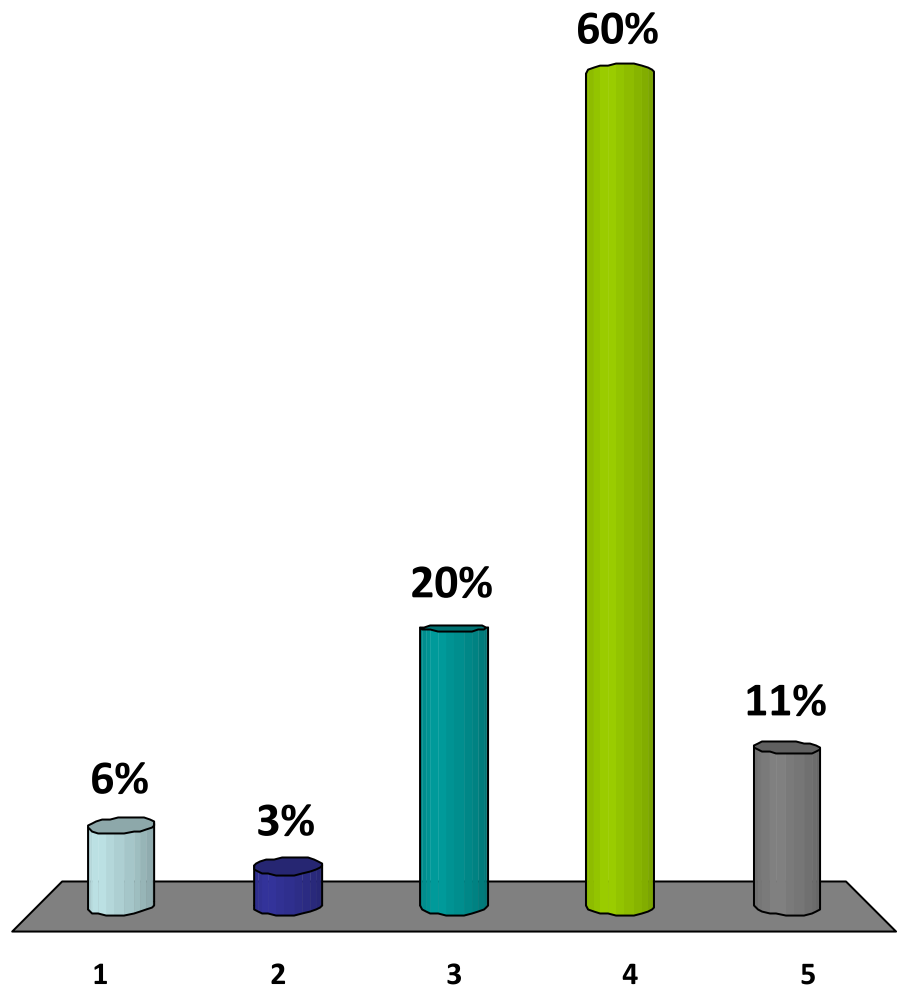
- Focus on logical parallelism
  - What steps can run in parallel?
  - What steps must be sequential?
- Machine independent
  - No fixed number of processors.
  - Assumes strong memory model.
- Relies on a scheduler to map algorithm to machine.
  - There exist provably good schedulers!

Example: Fibonacci

```
fib(n)
    if (n < 2) then
        return n;
    x = fib(n-1);
    y = fib(n-2);
    return x + y;
}
```

The running time of this algorithm is:

1. O(log n)
2. O(n)
3. O(n^2)
4. O(2^n)
5. I forget.



Response Counter

Just for fun…

Recall: matrix definition of Fibonacci Numbers

$$\begin{pmatrix} F_{k+2} \\ F_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix}$$

Induction:

$$\begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

How expensive to exponentiate 2x2 matrix? O(log k)

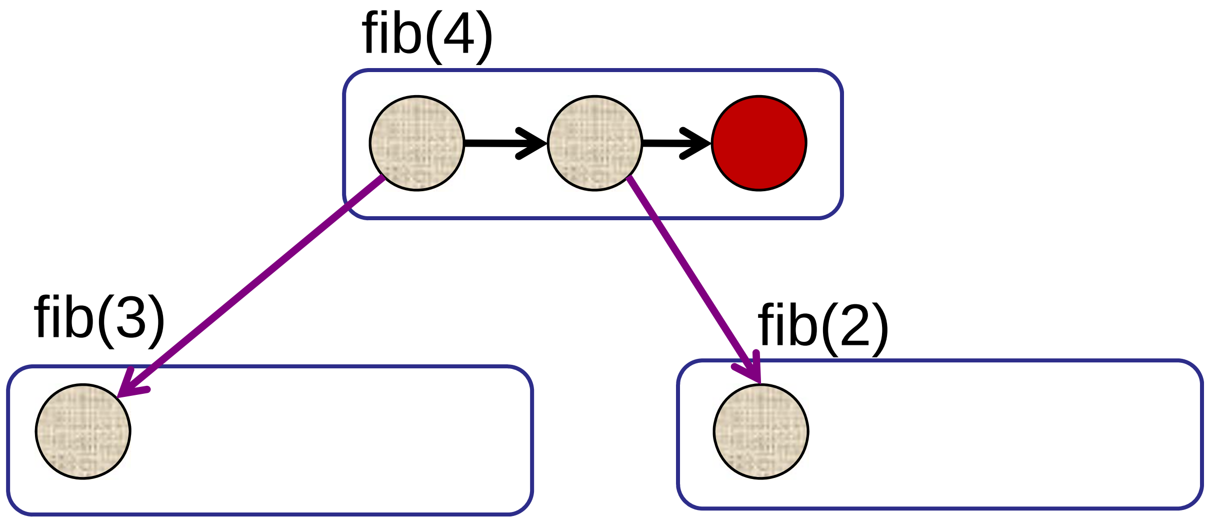
Example: Fibonacci

```
badFib(n)
  if (n < 2) then
        return n;
  x = badFib(n-1);
  y = badFib(n-2);
  return x + y;
}
```

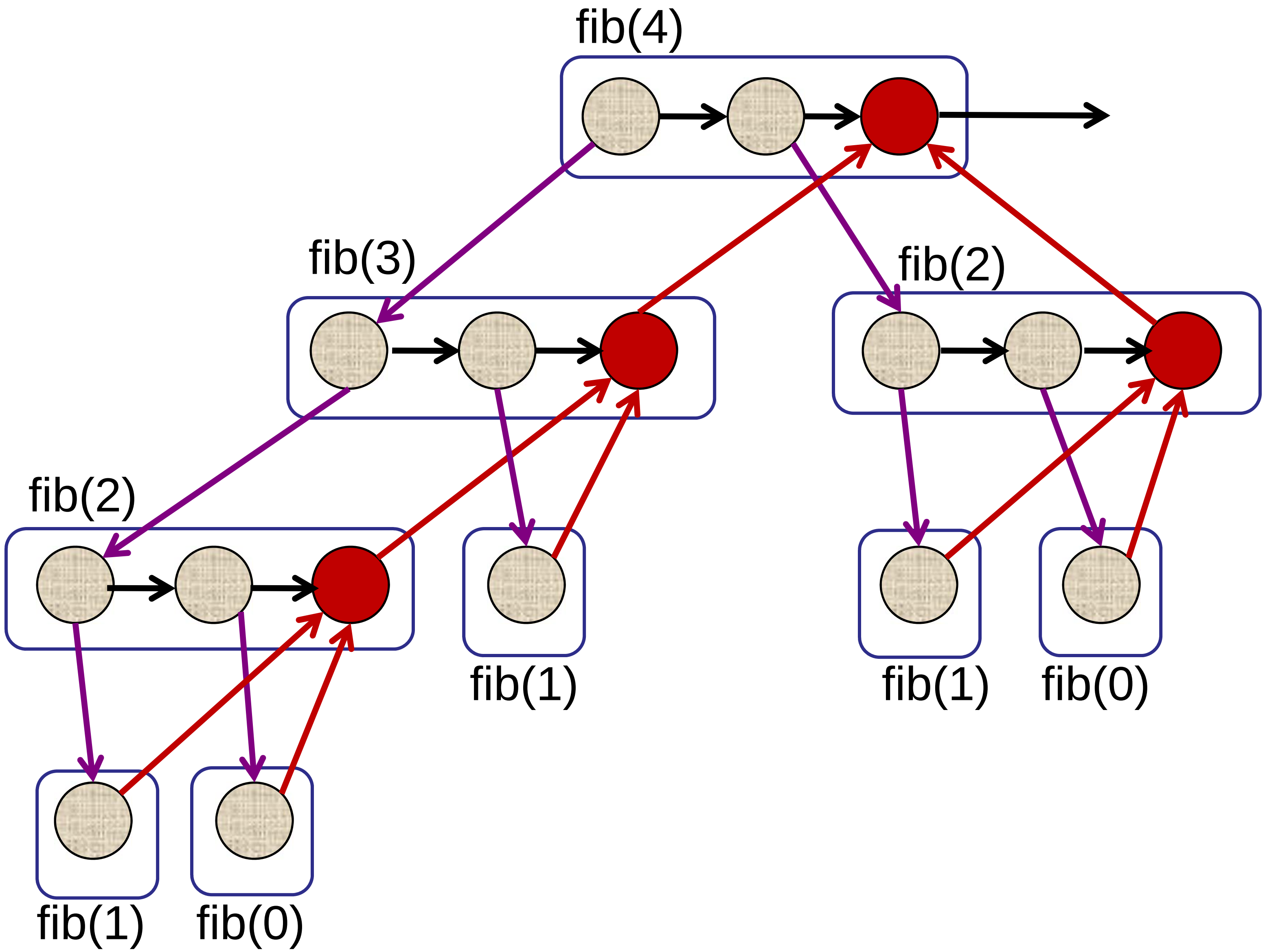
Example: Fibonacci

```
parallelFib(n)
  if (n < 2) then
       return n;
  x = spawn parallelFib(n-1);
  y = spawn parallelFib(n-2);
  sync;
  return x + y;
}
```

Executing Parallel Fibonacci



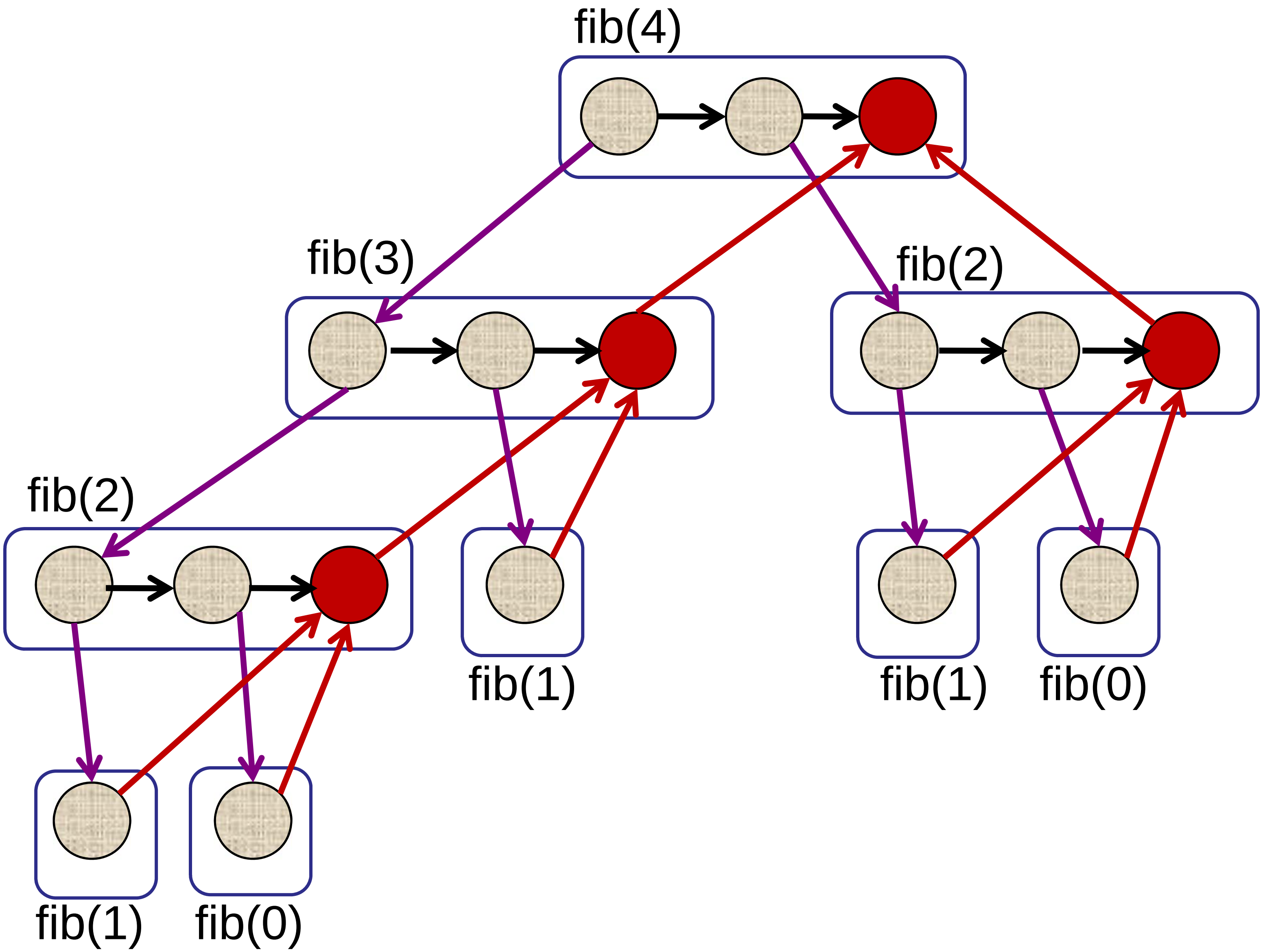
Executing Parallel Fibonacci



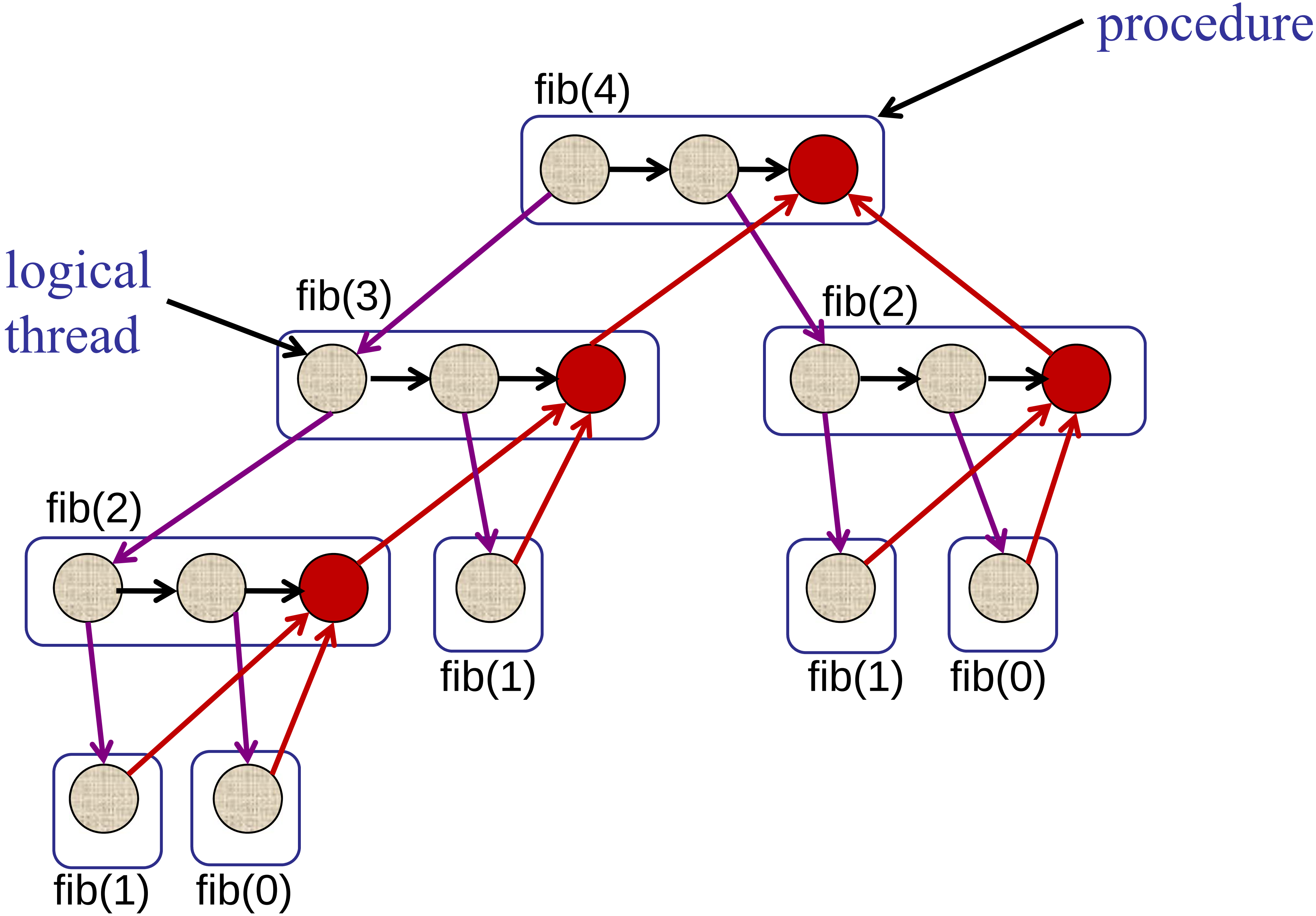
Example: Fibonacci

```
parallelFib(n)
  if (n < 2) then
     return n;
  x = spawn parallelFib(n-1);
  y = spawn parallelFib(n-2);
  sync;
  return x + y;
}
```

Executing Parallel Fibonacci



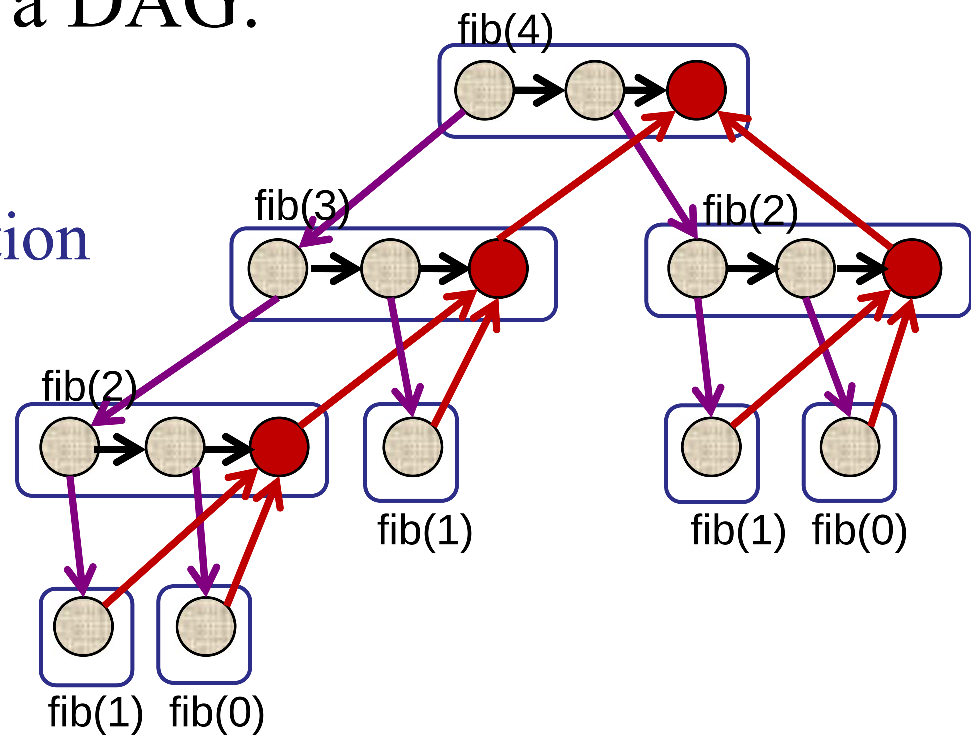
Executing Parallel Fibonacci



Parallel Computations

Represent a computation as a DAG:

- Directed acyclic graph
- Nodes = steps of computation
- Edges = dependencies
  - Spawn edges
  - Sync edges
  - Continuation edges



Scheduling a DAG-computation:

- On one processor…
- On many processors…

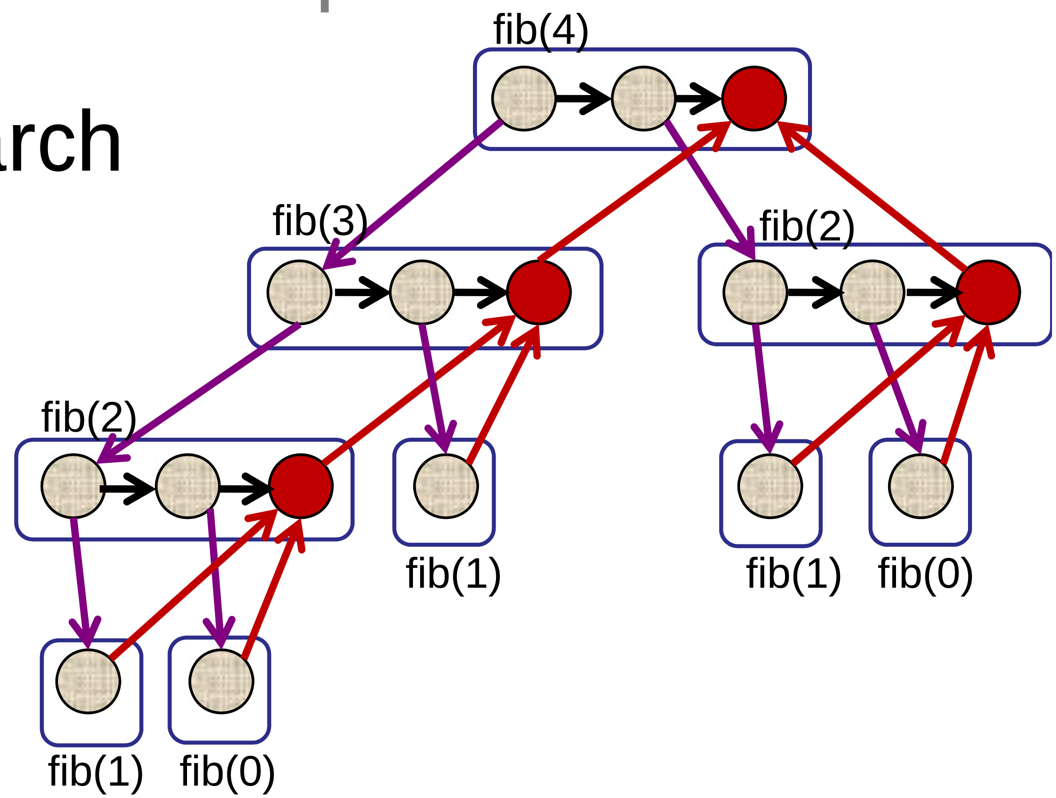
Simple model of parallel computation

Dynamic Multithreading

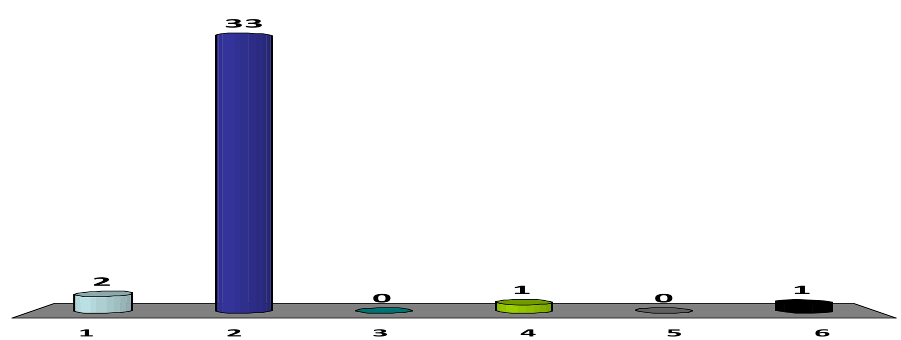
- Two special commands:
  - spawn: start a new (possibly parallel) procedure
  - sync: wait for all concurrent tasks to complete
- Machine independent
  - No fixed number of processors.
- Parallel computation modelled as DAG.
  - There exist provably good schedulers!

What algorithm could you use to schedule the DAG computation on 1 processor?

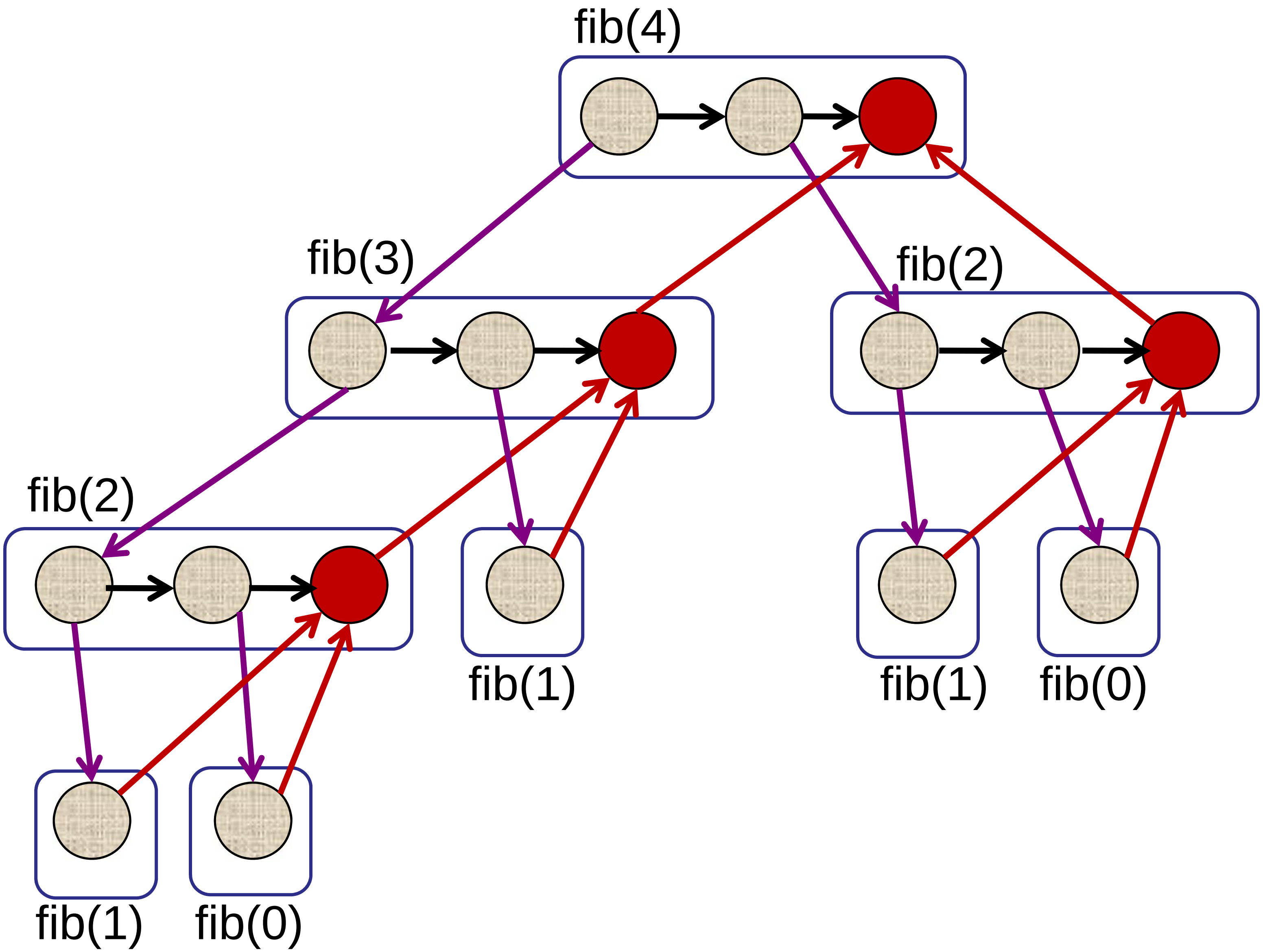
1. Breadth-First Search
2. Topological Sort
3. Bellman-Ford
4. Dijkstra's
5. Prim's
6. Something more complicated.



Response Counter

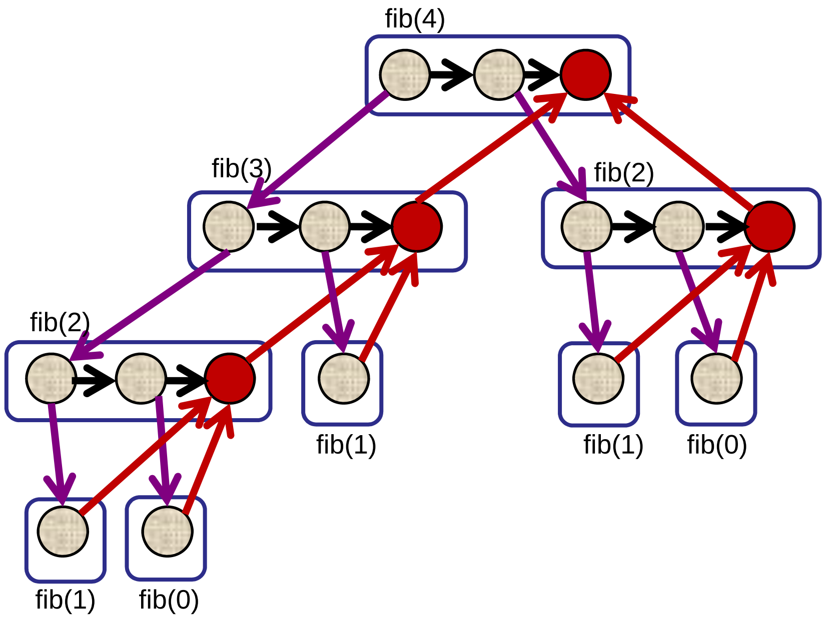


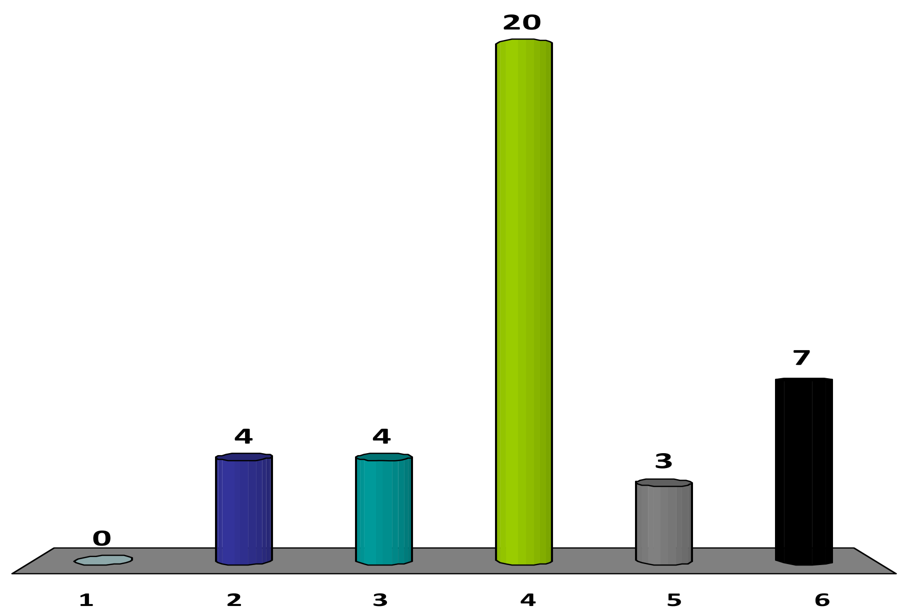
Schedule DAG on 1 processor?



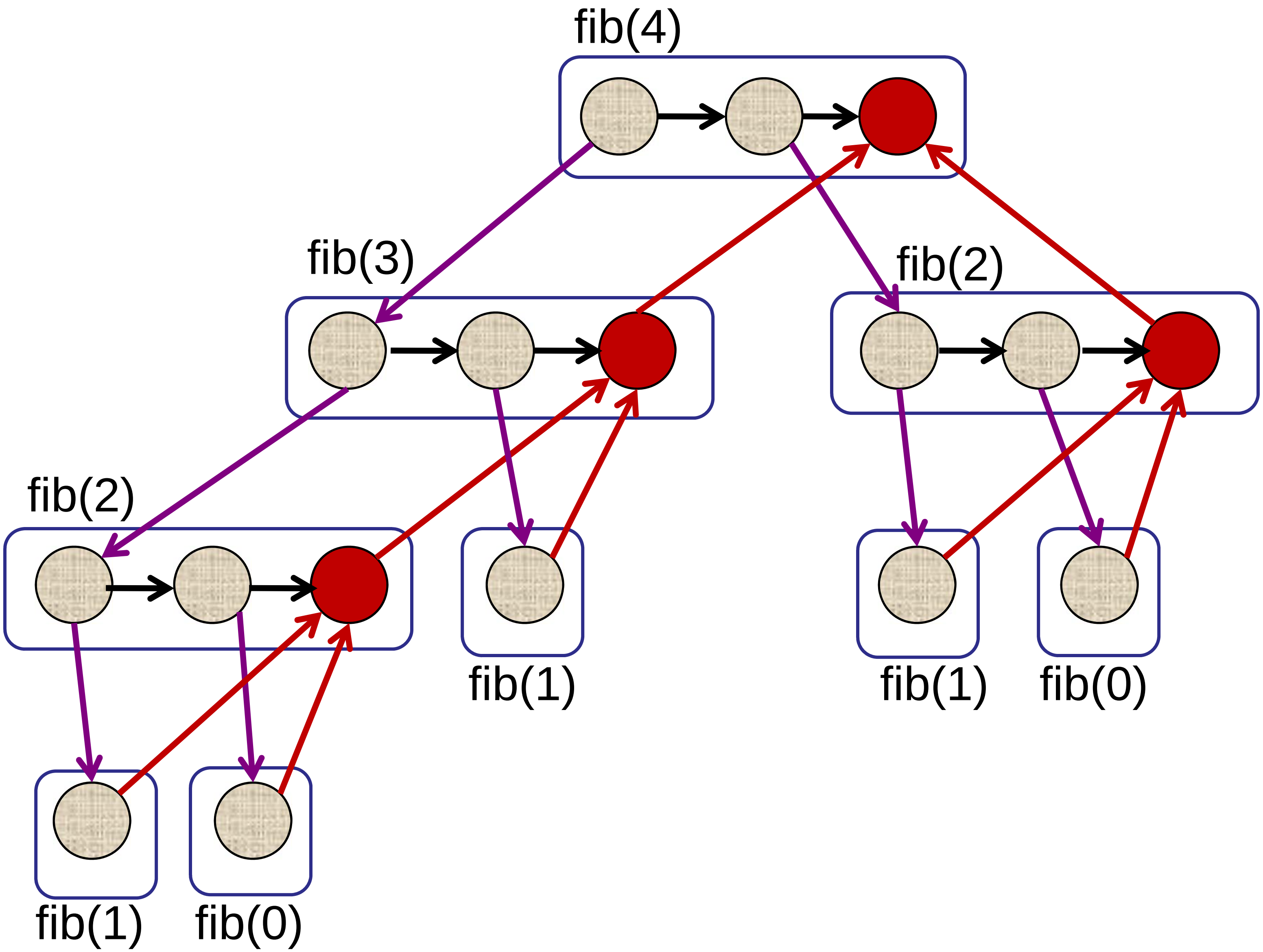
How many steps does fib(4) take if it is scheduled on **one** processor?

1. 1
2. 8
3. 13
4. 17
5. 32
6. I can't count.





How much work?



Calculating the Work

```
parallelFib(n)
  if (n < 2) then
     return n;
  x = spawn parallelFib(n-1);
  y = spawn parallelFib(n-2);
  sync;
  return x + y;
}
```

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Analyzing a Parallel Computation

Work: $\mathbf{T_1}$

- Total running time if executed on one processor.

  Equivalent: total steps taken on all processors.

- Calculate:
  - Count the number of nodes in your graph.
  - Set up a recurrence (as before).
  - Essentially, same as sequential algorithm analysis.

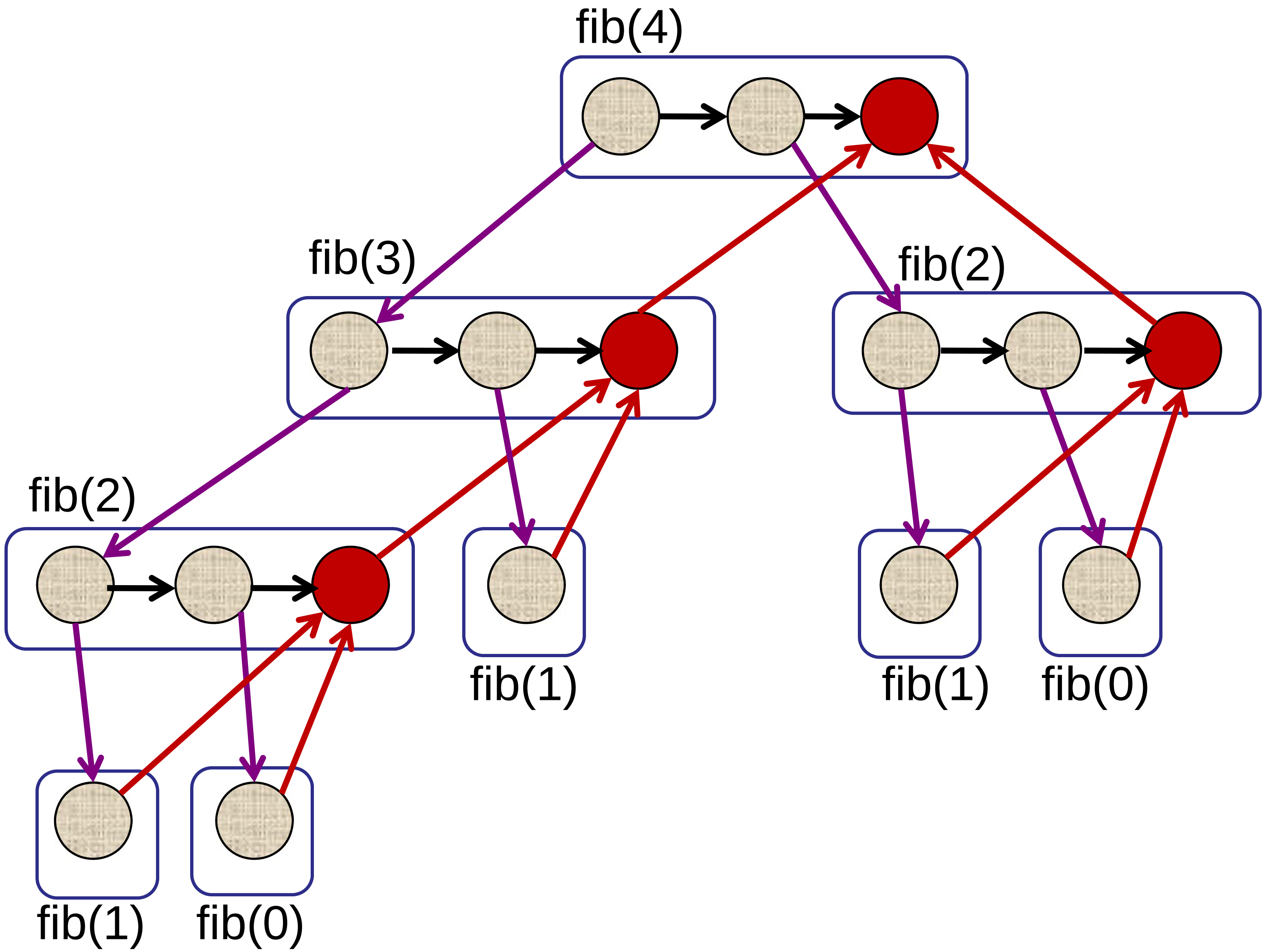
Analyzing a Parallel Computation

Definition: $\mathbf{T_p}$

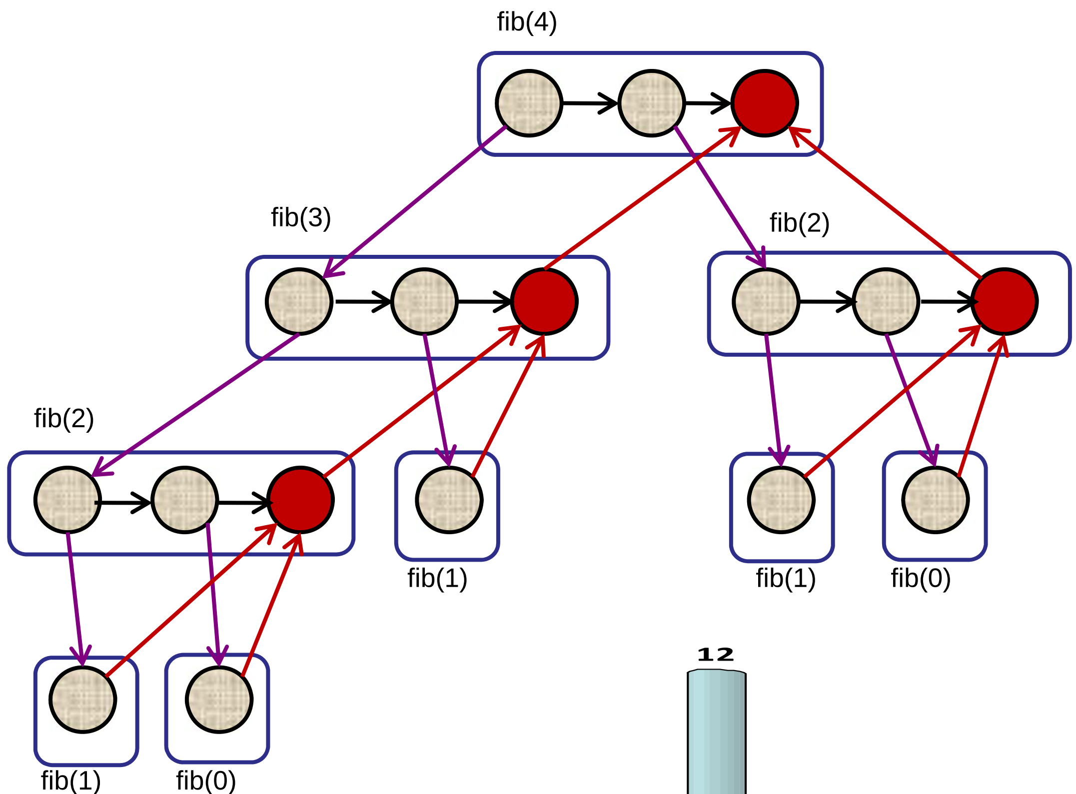
- Total running time if executed on ***p*** processors.

- Calculate:
  - Depends on the scheduler.
  - Cannot be easily calculated for arbitrary *p*.
  - We will see how to bound $\mathbf{T_p}$ given a *good* scheduler.

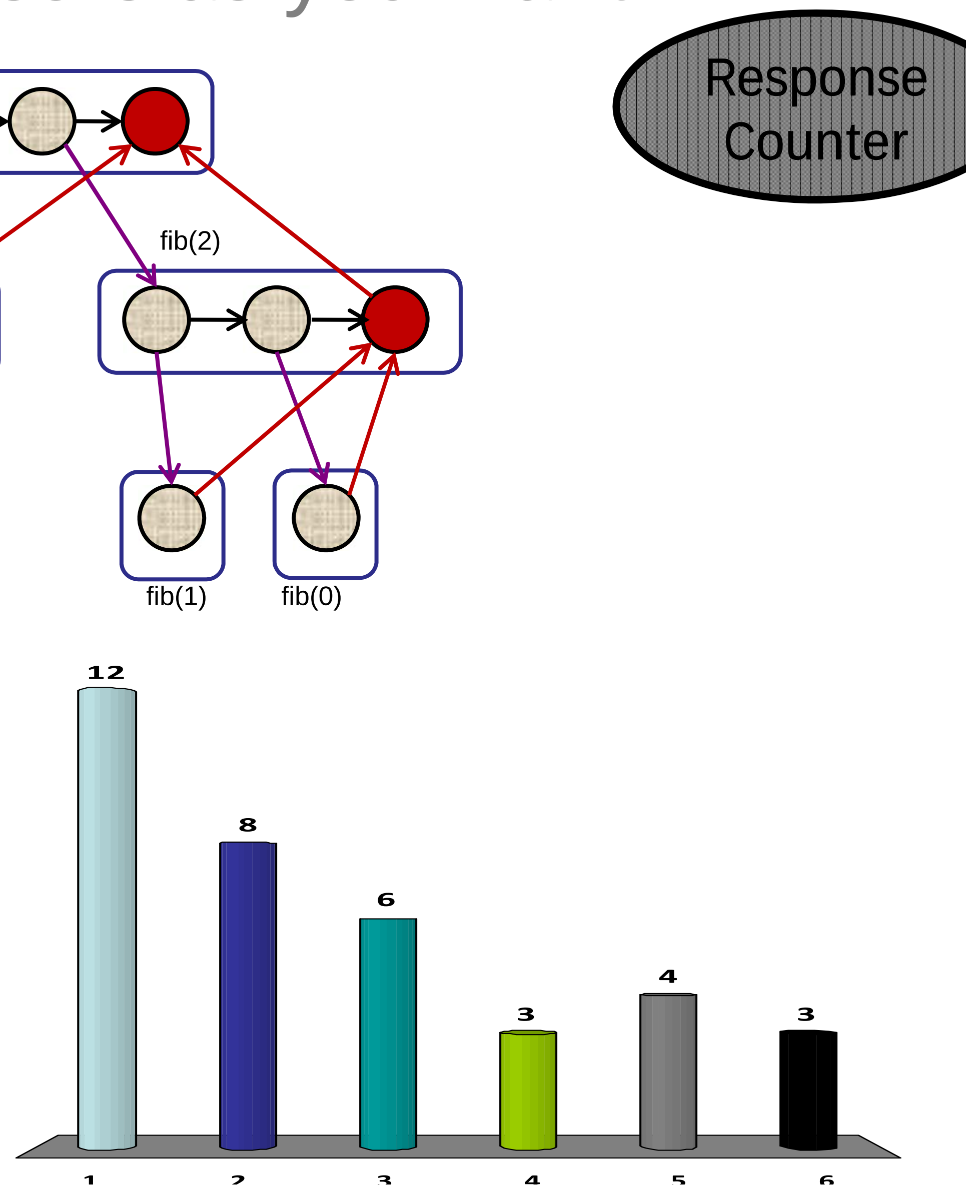
Schedule DAG on ∞ processors?



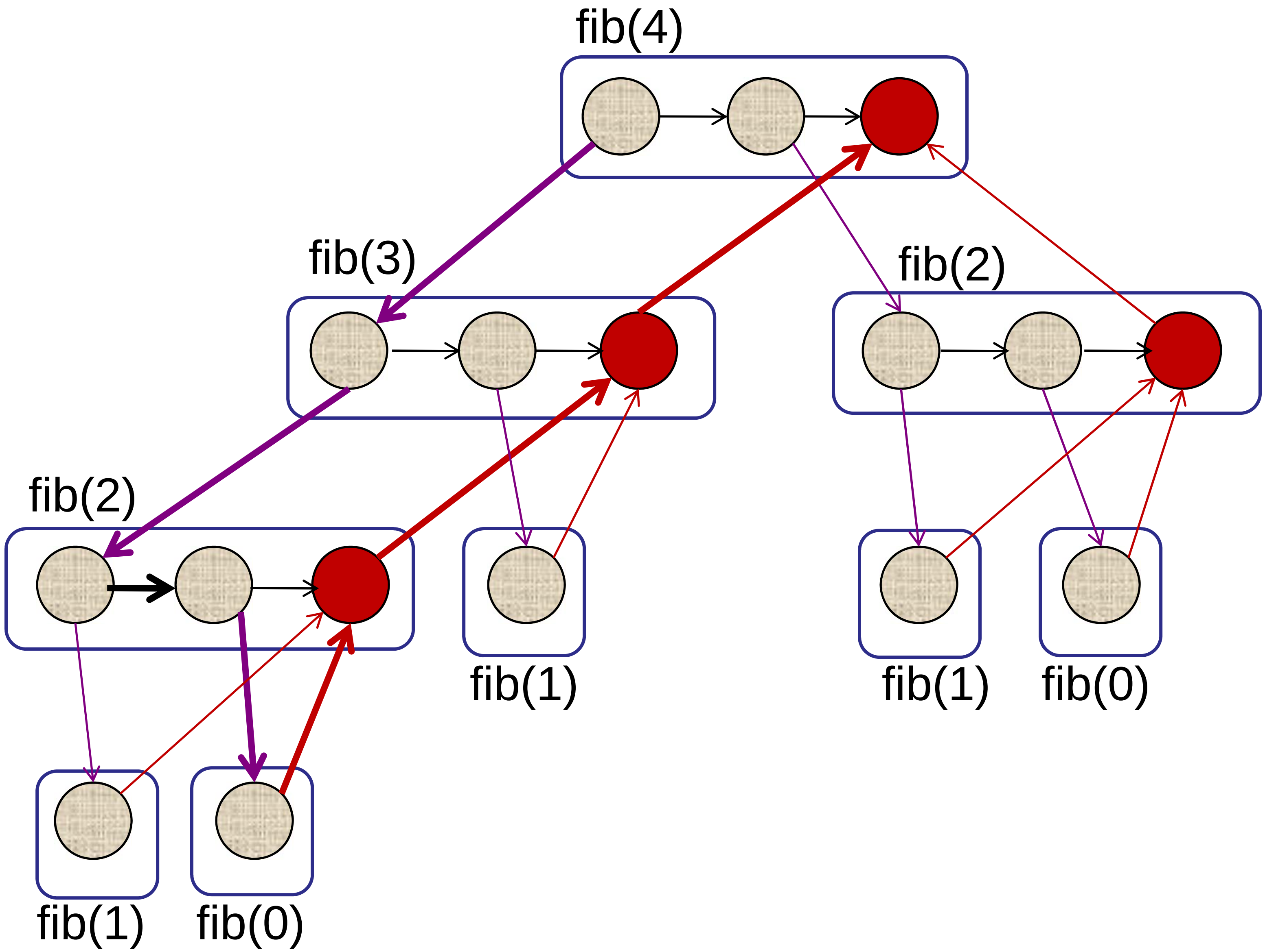
What is the **fastest** fib(4) can run if you have as many processors as you want?



1. 4
2. 6
3. 7
4. 8
5. 10
6. I forgot to count.



Schedule DAG on ∞ processors?



Analyzing a Parallel Computation

Critical Path: $\mathbf{T}_{\infty}$

- Total running time if executed on infinite processors.

  Equivalent: what is the *fastest* the program can execute?

- Calculate:
  - Find the longest path in the DAG.
  - Set up a recurrence….

Example: Fibonacci

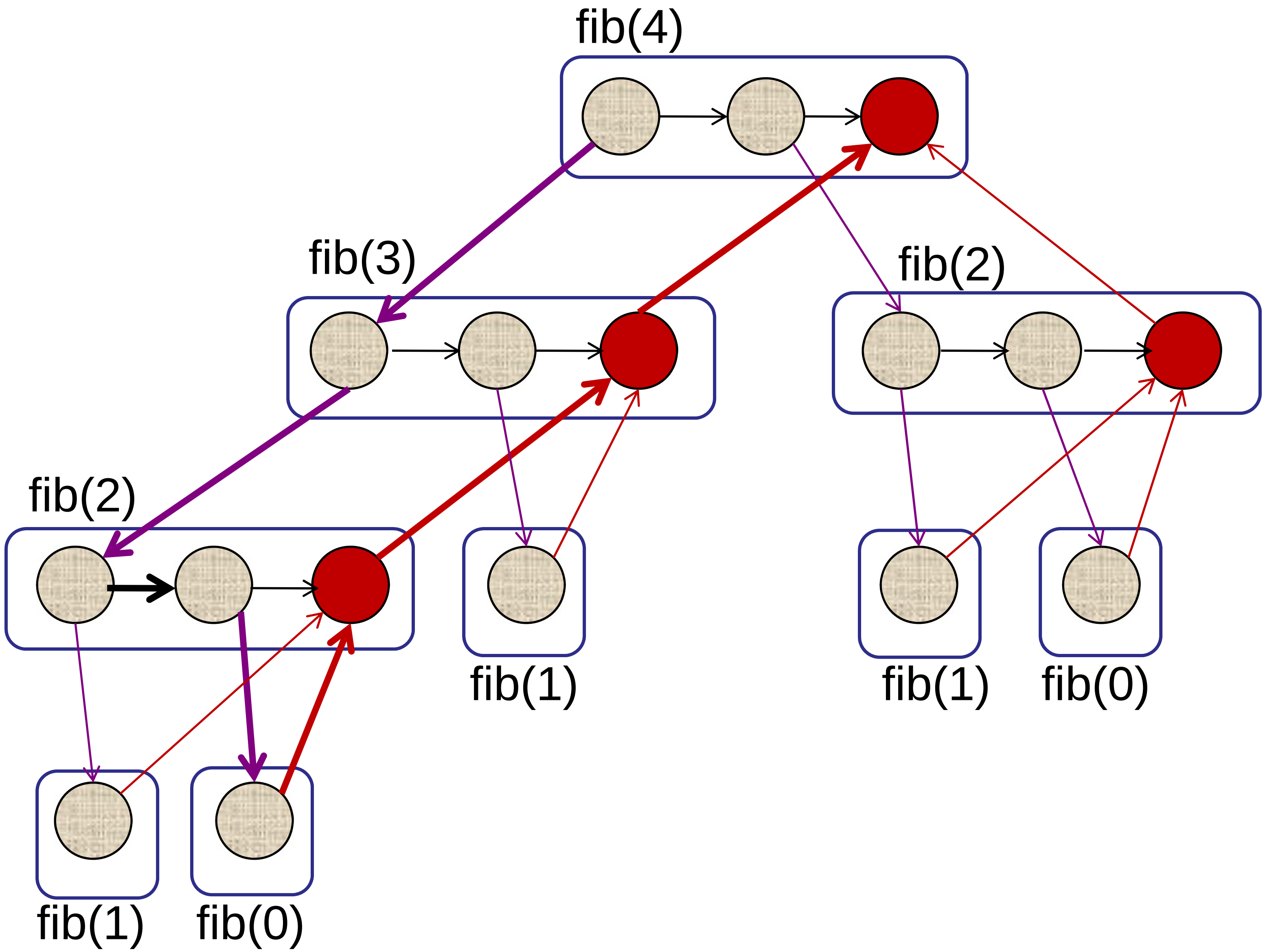
```
parallelFib(n)
  if (n < 2) then
     return n;
  x = spawn parallelFib(n-1);
  y = spawn parallelFib(n-2);
  sync;
  return x + y;
}
```

MAX

+

$$T_\infty(n) = \max(T_\infty(n-1)\,,\,T_\infty(n-2)) + O(1)$$

Schedule DAG on ∞ processors?



Example: Fibonacci

```
parallelFib(n)
  if (n < 2) then
     return n;
  x = spawn parallelFib(n-1);
  y = spawn parallelFib(n-1);
  sync;
  return x + y;
}
```

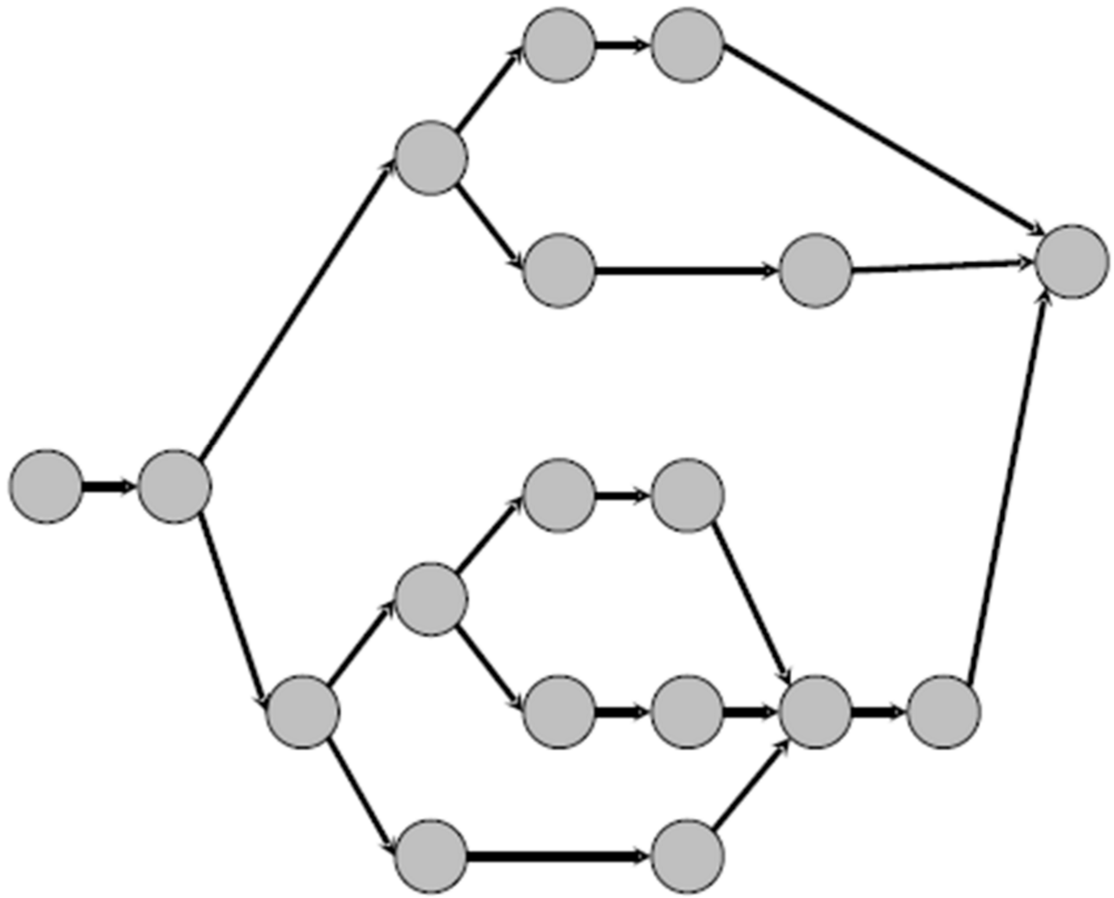
$$T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + O(1)$$

$$= O(n)$$

Analyzing Parallel Algorithms

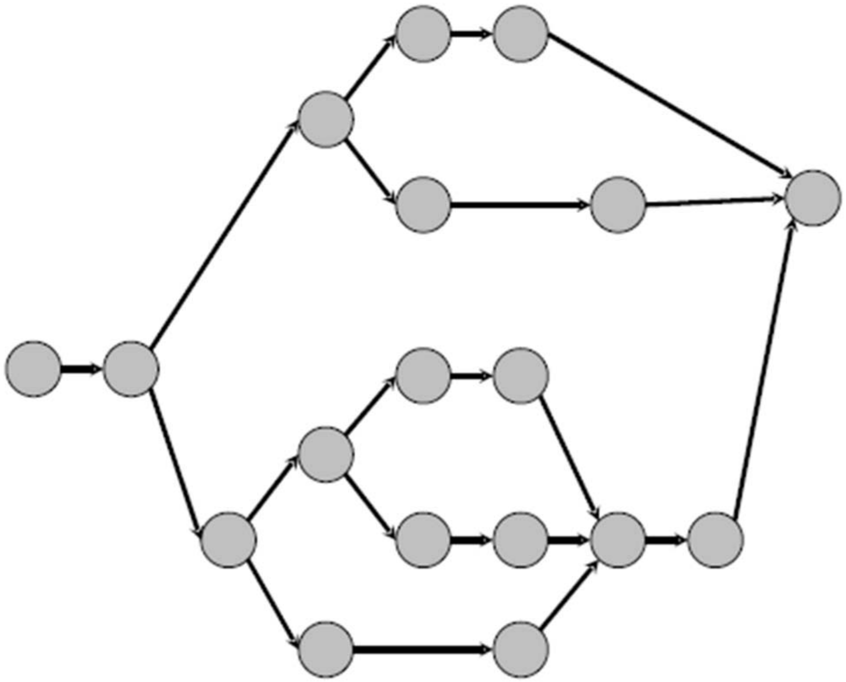
Key metrics:

- Work: T_1
- Critical Path: T_∞

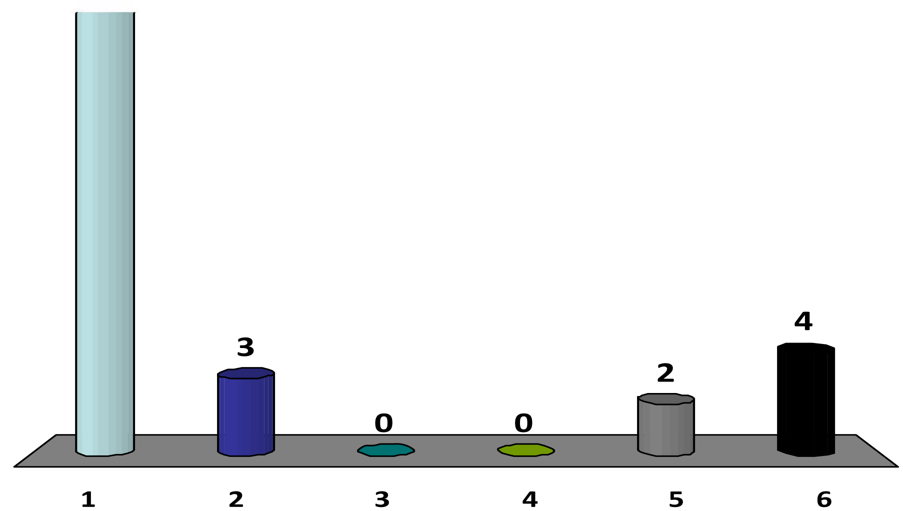


What is the work?

What is the work on this graph?



1. 18
2. 24
3. 32
4. 37
5. 45
6. I forgot to count.

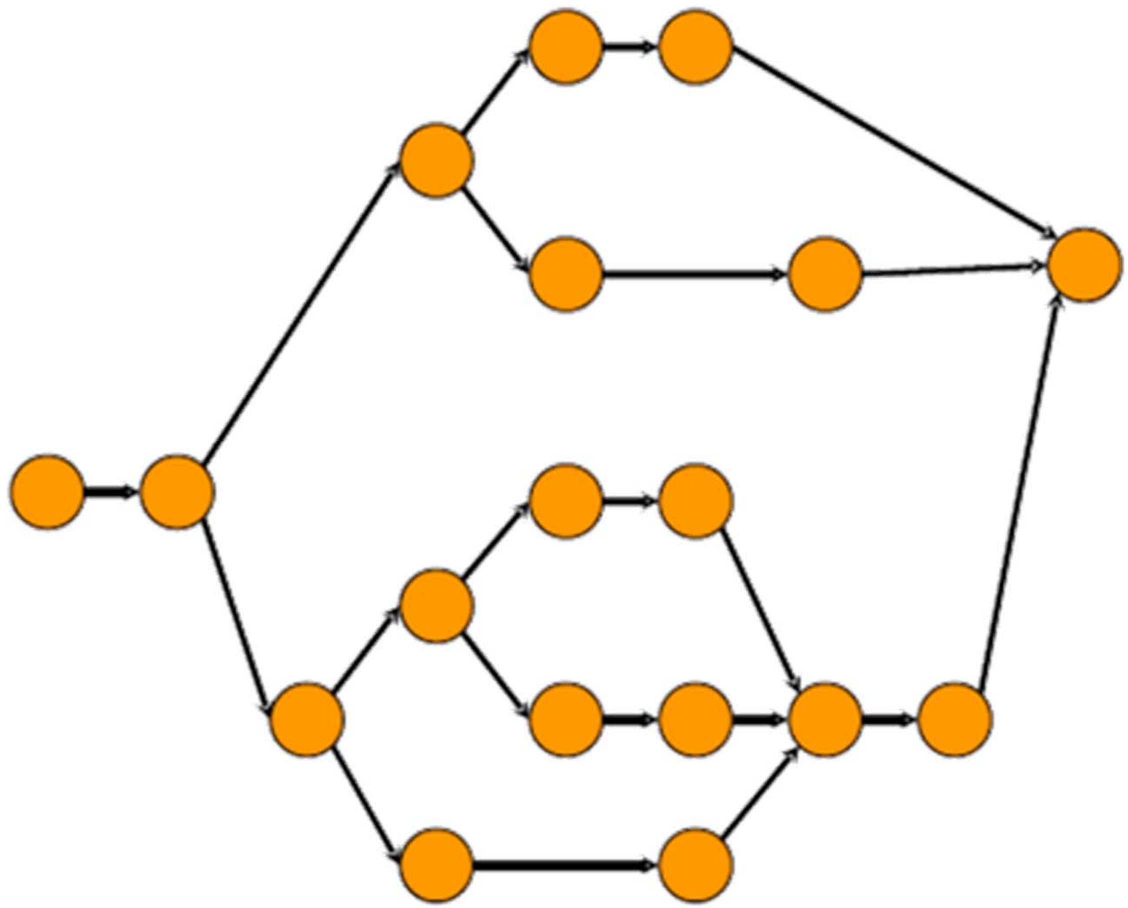


Response Counter

Analyzing Parallel Algorithms

Key metrics:

- Work: T_1
- Critical Path: T_∞

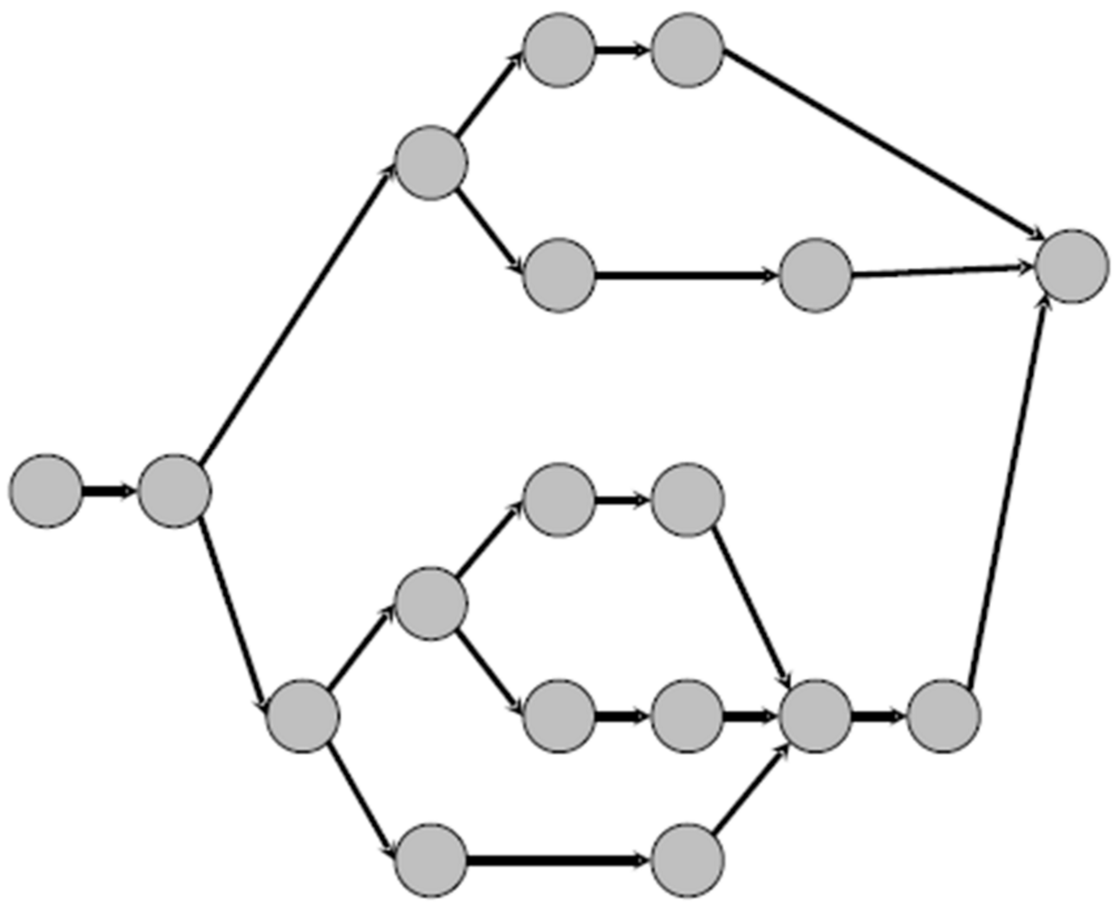


Work = 18

Analyzing Parallel Algorithms

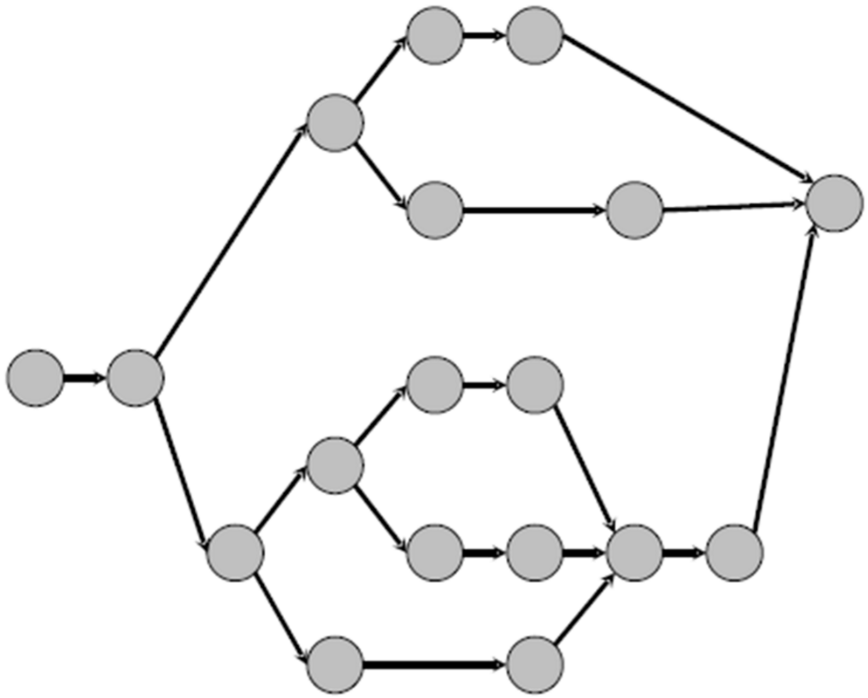
Key metrics:

- Work: T_1
- Critical Path: T_∞

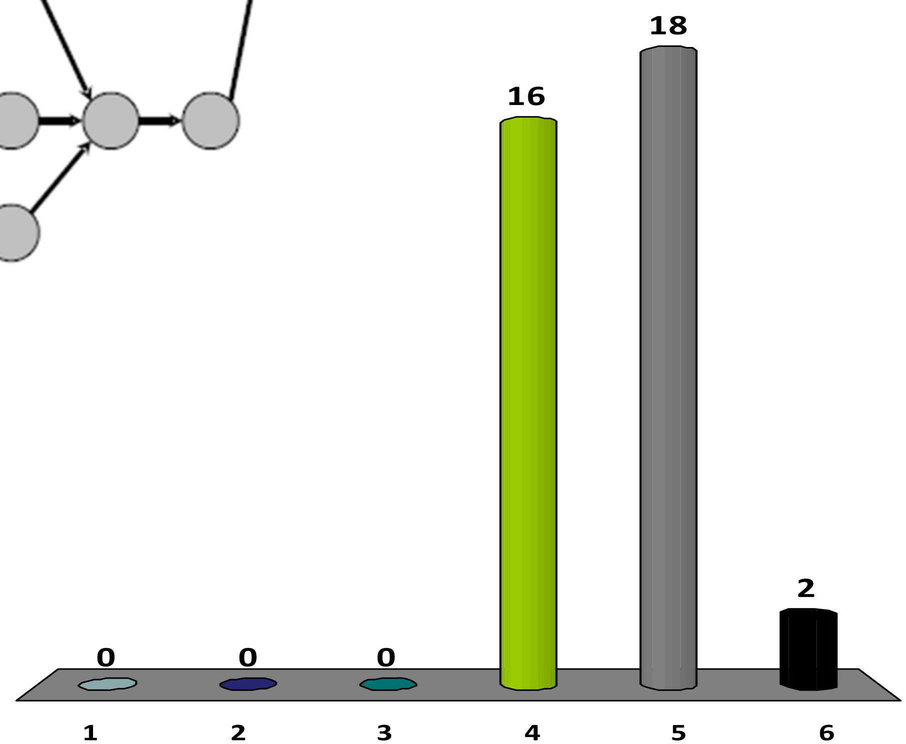


What is the critical path on this graph?

1. 5
2. 6
3. 7
4. 8
5. 9
6. I forgot to count.



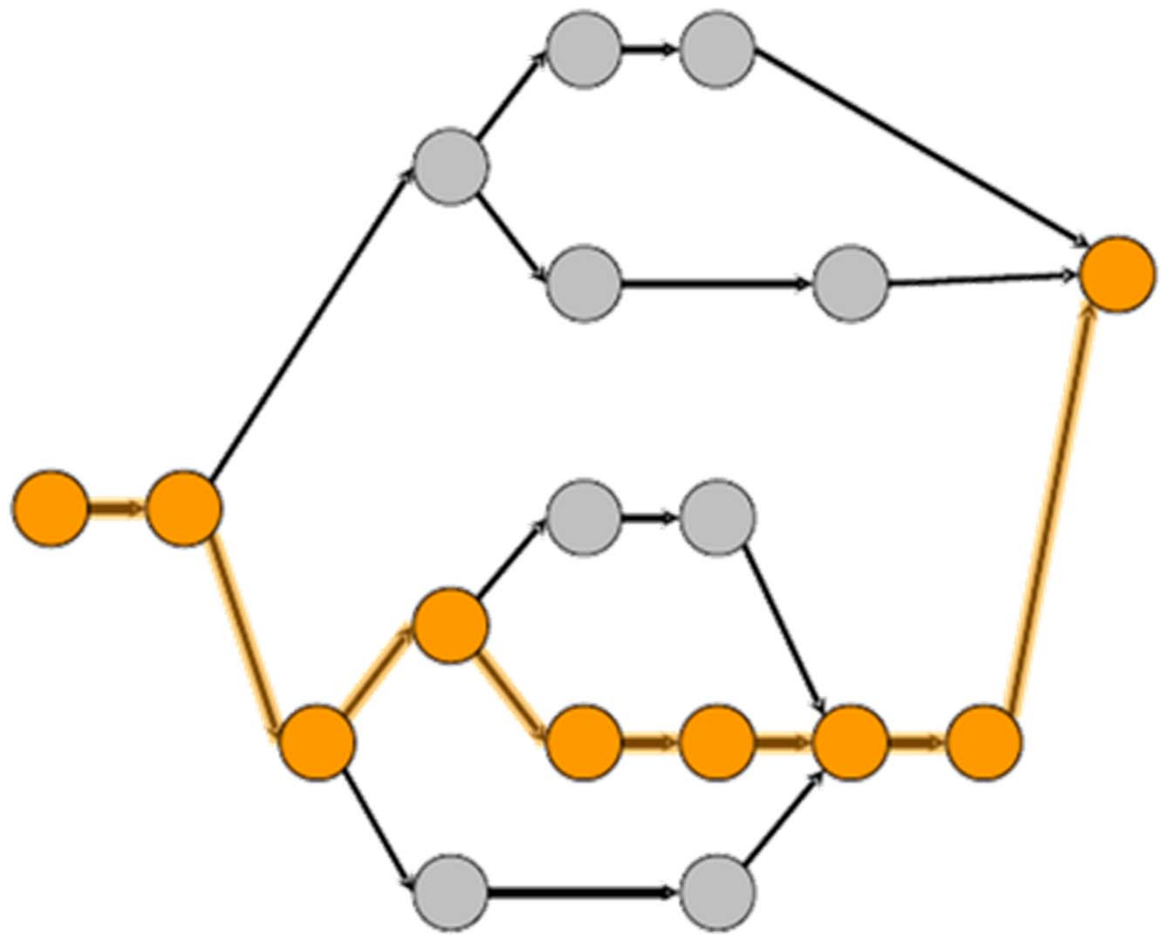
Response Counter



Analyzing Parallel Algorithms

Key metrics:

- Work: T_1
- Critical Path: T_∞



Critical path = 9

Analyzing Parallel Algorithms

Parallelism: How parallel is your program?

Analyzing Parallel Algorithms

Parallelism: How parallel is your program?

- How many processes does it scale to?
- How many processes can we usefully use?

Analyzing Parallel Algorithms

Key metrics:

- Work: T_1
- Critical Path: T_∞

Parallelism:

- How parallel is your program?
- Example: original (non-parallel) badFibonacci
  - $T_1 = O(2^n)$
  - $T_\infty = O(2^n)$
  - Parallelism $= (T_1 / T_\infty) = 1$

Analyzing Parallel Algorithms

Key metrics:

- Work: T_1
- Critical Path: T_∞

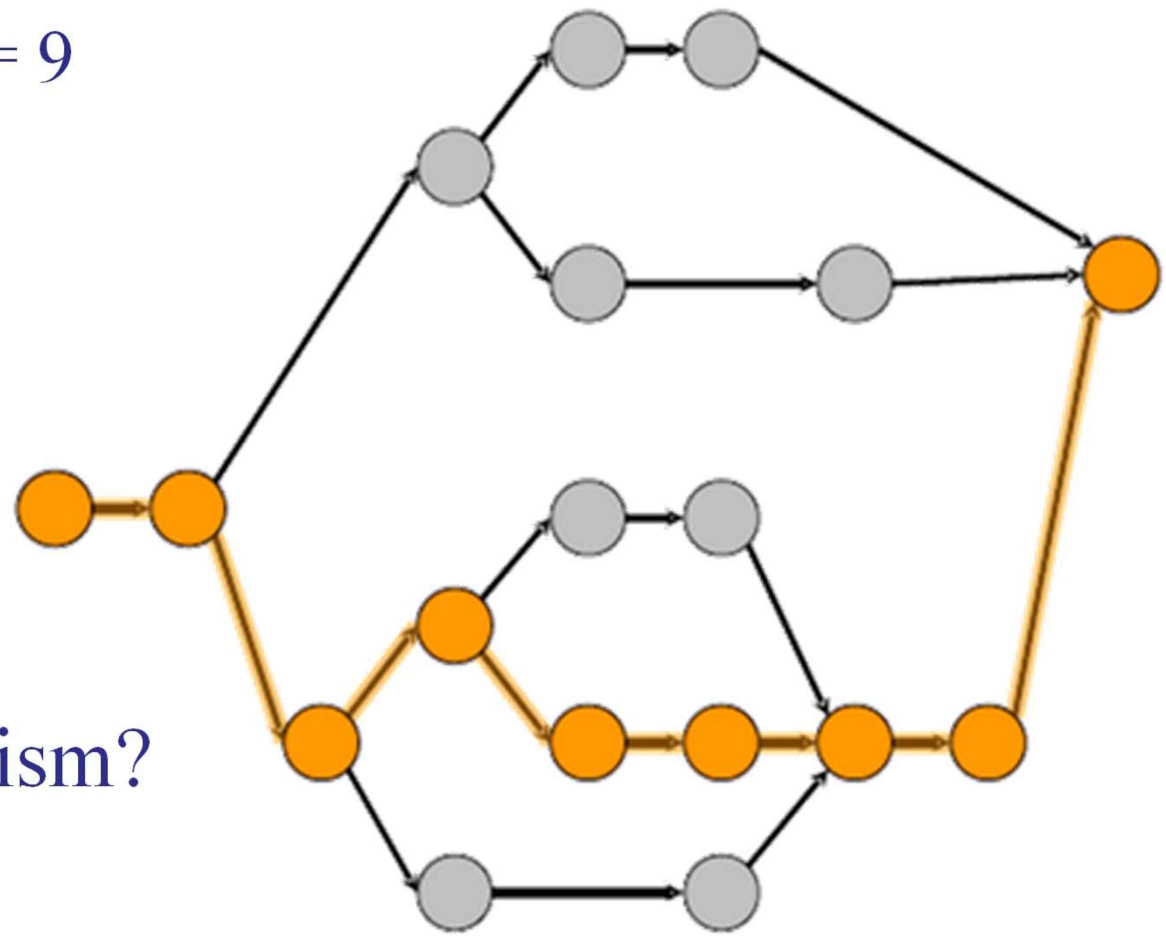
Parallelism:

- How parallel is your program?
- Example: parallel Fibonacci
  - $T_1 = O(2^n)$
  - $T_\infty = O(n)$
  - Parallelism = $(T_1 / T_\infty) = O(2^n / n)$

Analyzing Parallel Algorithms

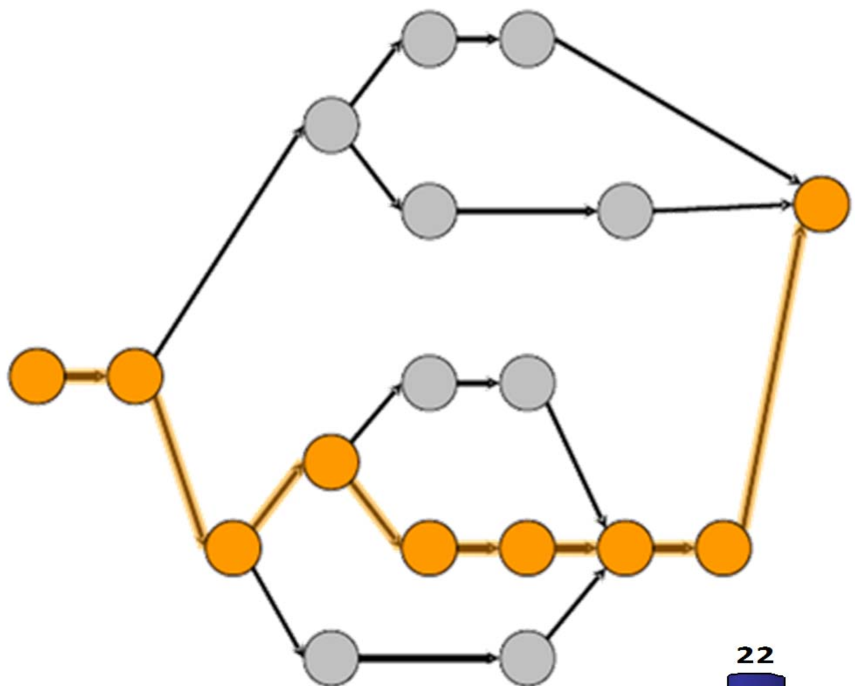
Key metrics:

- Work: $T_1 = 18$
- Critical Path: $T_\infty = 9$

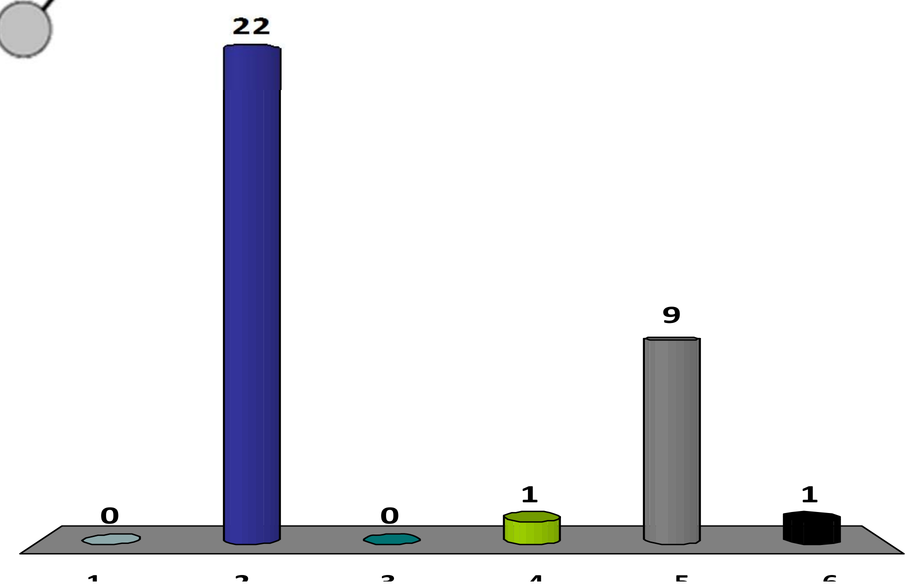


What is the parallelism?

What is the parallelism of this graph?



1. 1
2. 2
3. 3
4. 4
5. 5
6. I forgot to count.

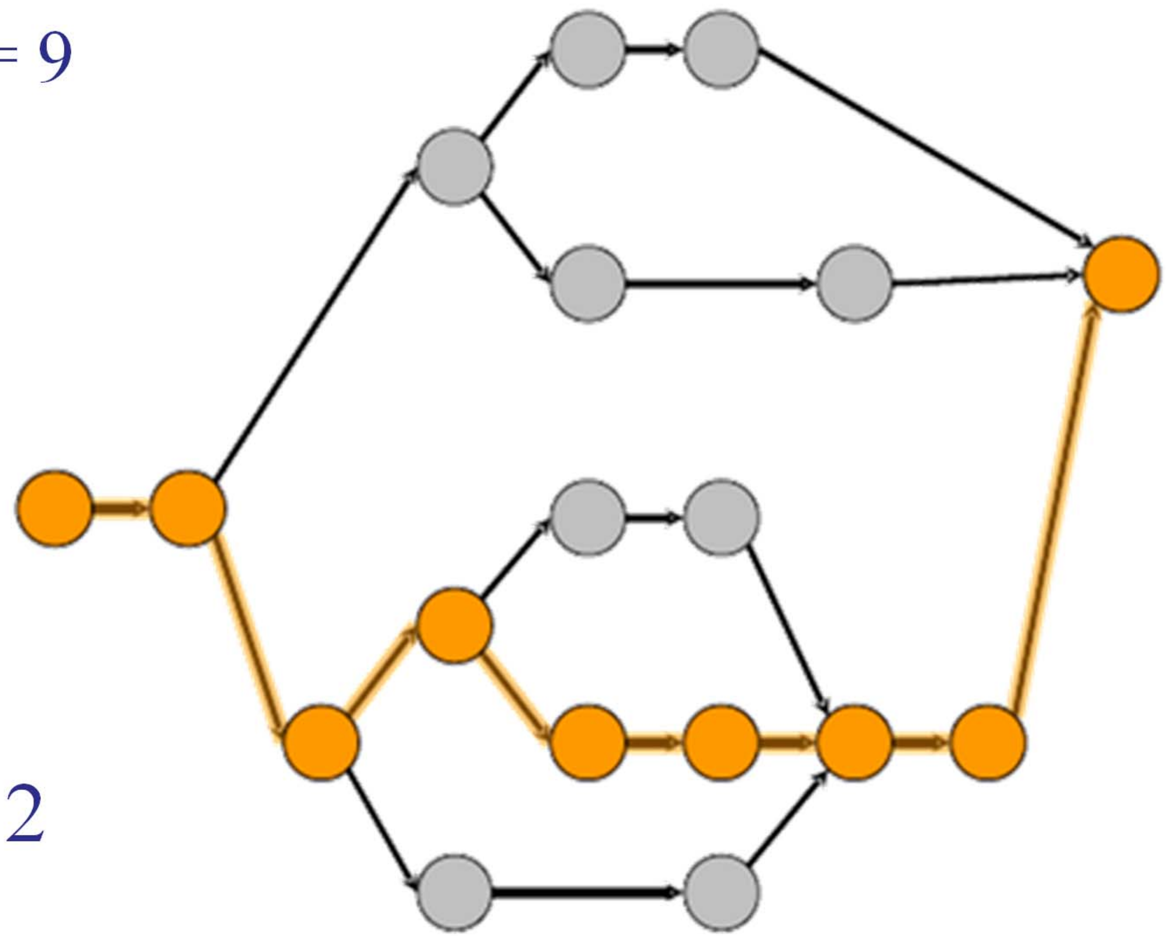


Response Counter

Analyzing Parallel Algorithms

Key metrics:

- Work: $T_1 = 18$
- Critical Path: $T_\infty = 9$



Parallelism = 18/9 = 2

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: $\mathbf{T_p}$

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

- Goal: $\mathbf{T_p = (T_1 / p) + T_\infty}$
  - Almost optimal (within a factor of 2).
  - We have to spend time $\mathbf{T_\infty}$ on the critical path.

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

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

Analyzing Parallel Algorithms

Key metrics:

- Work: T_1
- Critical Path: T_∞
- Parallelism: (T_1 / T_∞)

Running time on **p** processors:

- Assume $p = (T_1 / T_\infty)$
- Then:

$$\mathbf{T_p = (T_1 / p) + T_\infty = 2T_\infty}$$

Analyzing a Parallel Computation

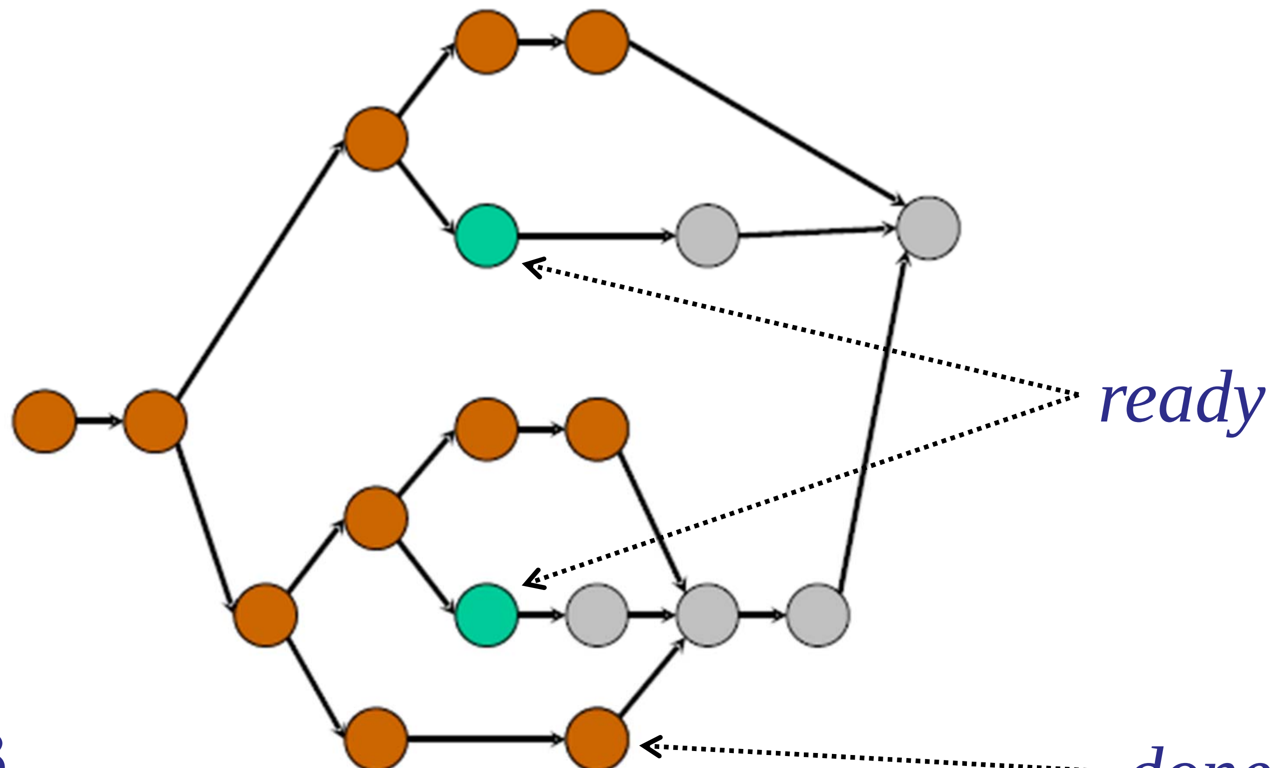
Greedy Scheduler

- 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.

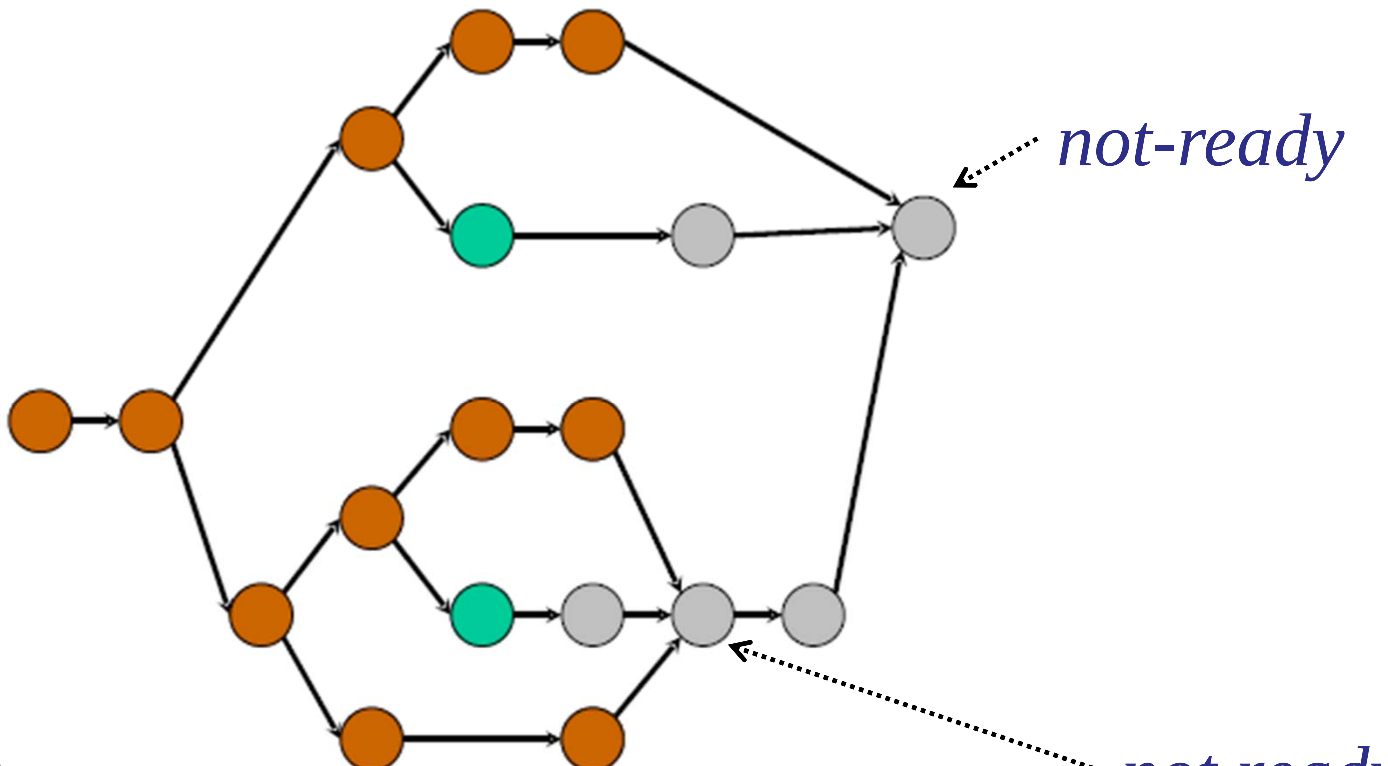


Assume $p = 3$

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.

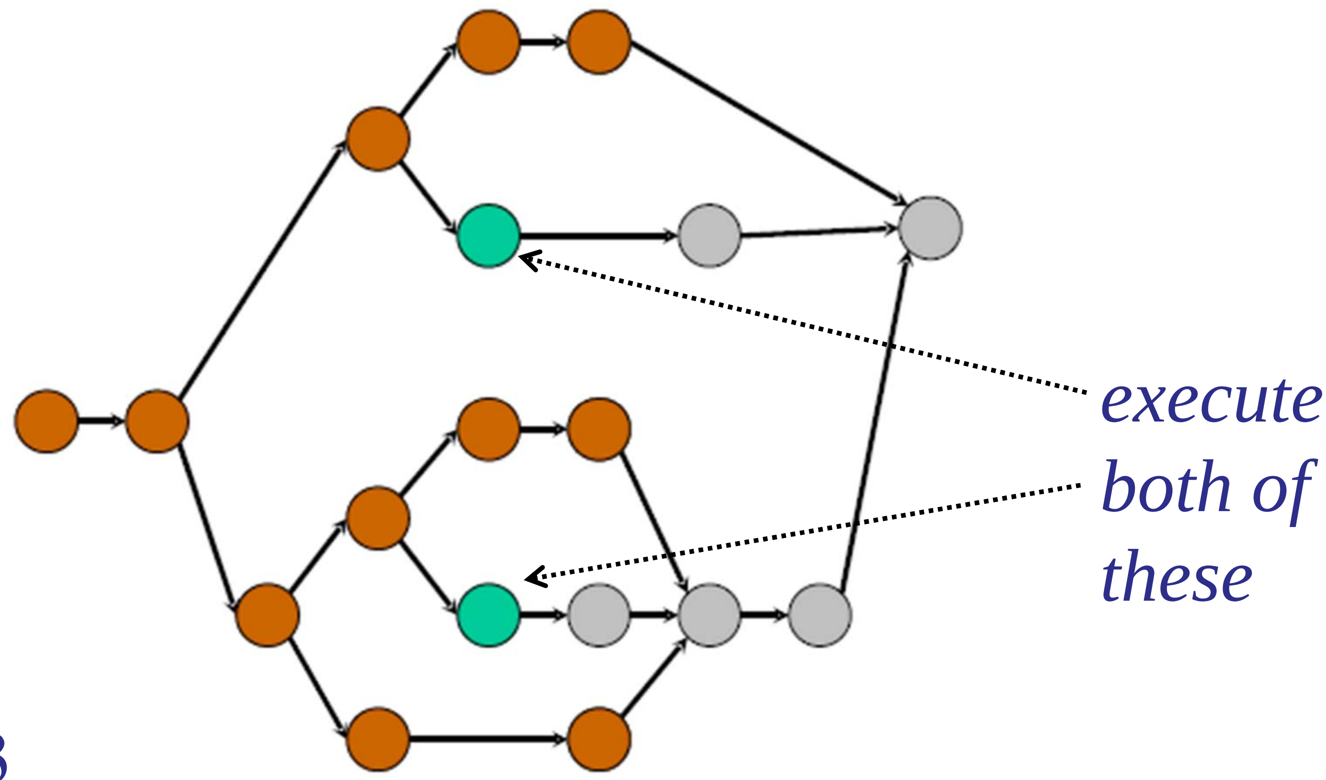


Assume $p = 3$

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.

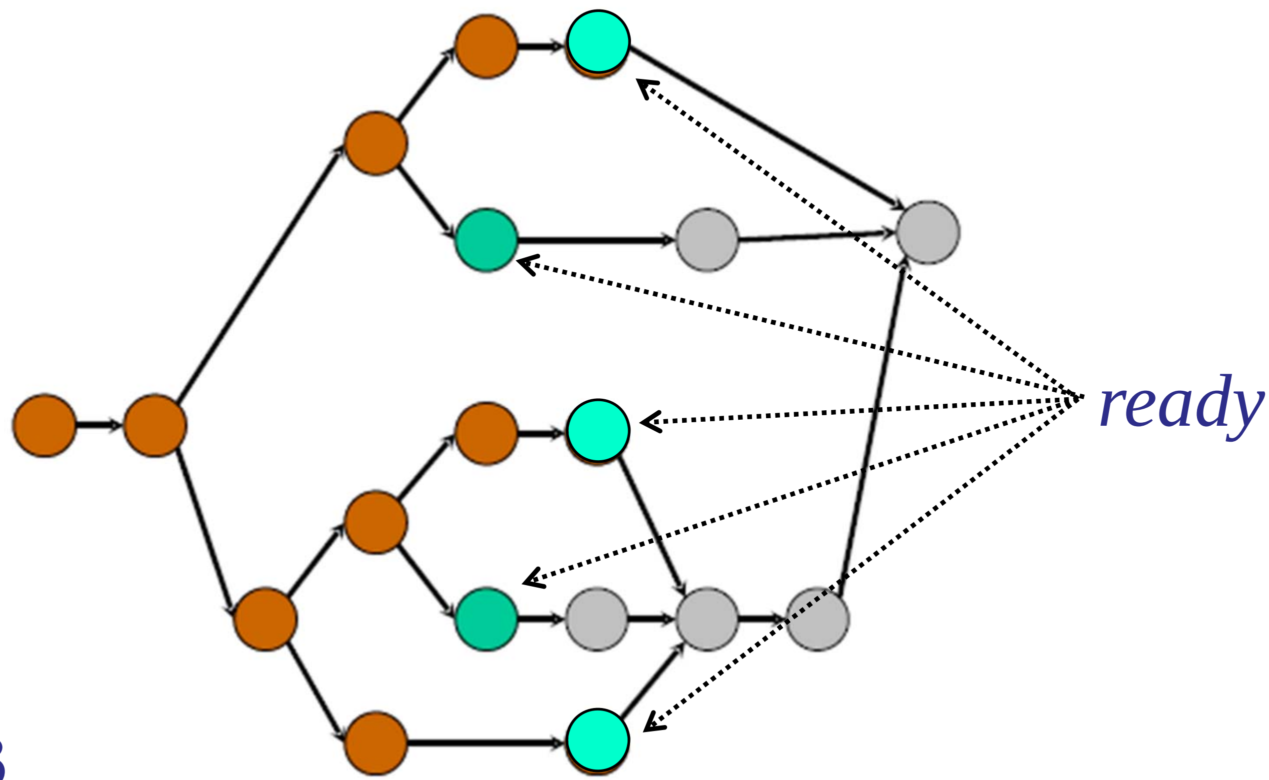


Assume $p = 3$

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.

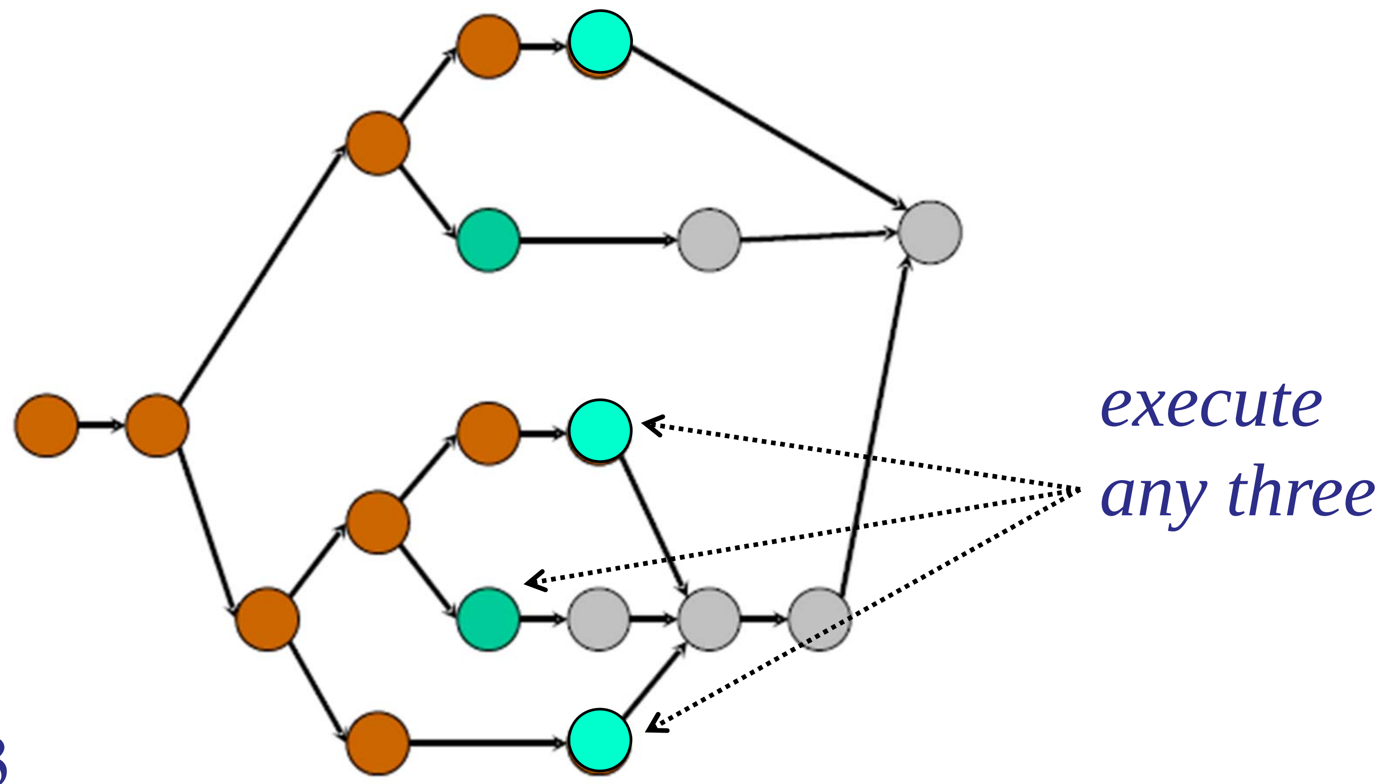


Assume $p = 3$

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., the Cilk / Cilk++ scheduler.
- Now part of Intel Parallel Studio

Parallel Analysis… a story

Socrates:

- a parallel chess program from ~1995
- used to demonstrate advantage of work-stealing
- defeated an early version of Deep Blue (the chess computer that eventually beat Kasparov)

Parallel Analysis… a story

Socrates:

- a parallel chess program from ~1995
- used to demonstrate advantage of work-stealing
- defeated an early version of Deep Blue (the chess computer that eventually beat Kasparov)

- Development: 32-processor machine
- Deployed (for competition): 512-processor machine

Parallel Analysis… a story

Socrates:

	Original Version	New Version
T_{32}	65 sec.	40 sec.

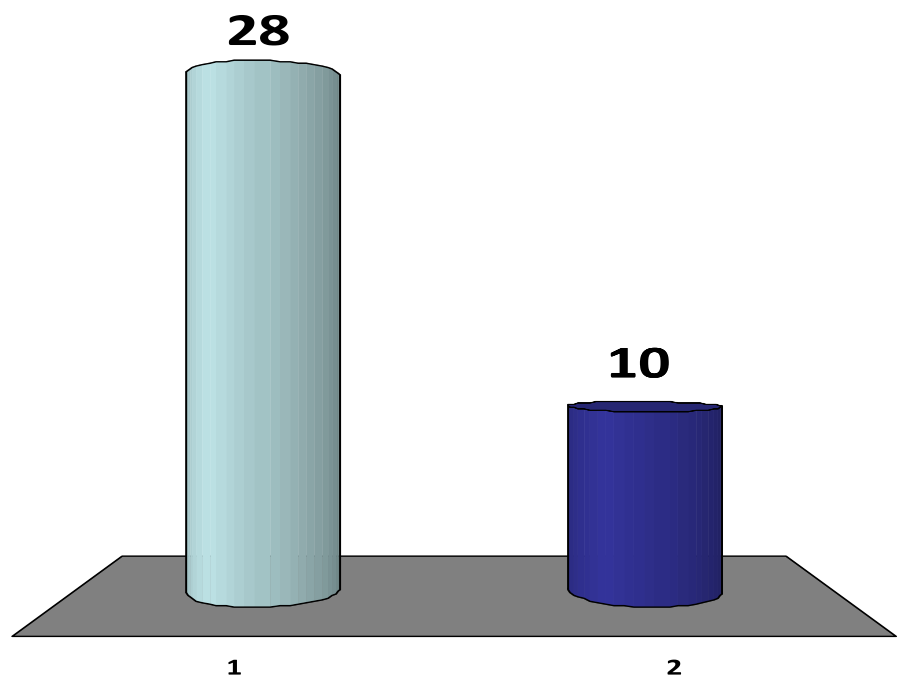
Parallel Analysis… a story

Socrates:

	Original Version	New Version
T_{32}	65 sec.	40 sec.
T_1	2048 sec.	1024 sec.
T_∞	1 sec.	8 sec.

Which is better?

1. Original version
2. New version



	Original Version	New Version
T_{32}	65 sec.	40 sec.
T_1	2048 sec.	1024 sec.
T_∞	1 sec.	8 sec.

Response Counter

Parallel Analysis… a story

Socrates:

	Original Version	New Version
T_{32}	65 sec.	40 sec.
T_1	2048 sec.	1024 sec.
T_∞	1 sec.	8 sec.
T_{512}	$T_1/512 + T_\infty$	$T_1/512 + T_\infty$
	$2048/512 + 1$	$1024/512 + 8$
	5 sec.	10 sec.

So far…

- Model for parallel algorithms
  - Dynamic multithreading
  - spawn/synch semantics

- Metrics for analyzing parallel programs
  - Work
  - Critical path
  - Parallelism
  - Speed-up

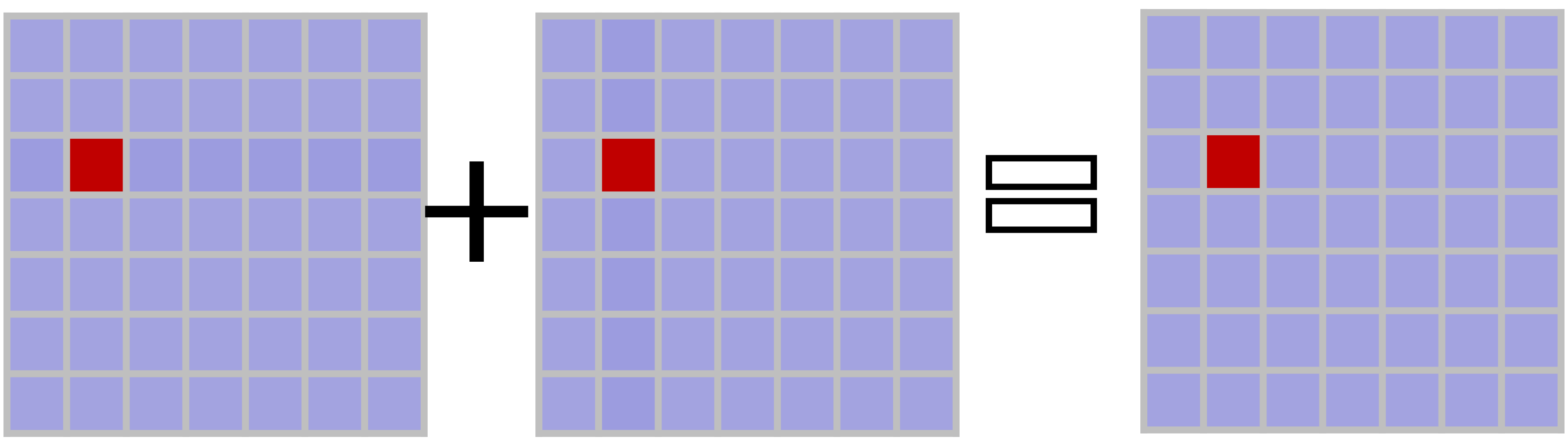
Matrix Addition

Add(A,B)

    for i = 1 **to** n **do**

        for j = 1 **to** n **do**

            C[i,j] = A[i,j] + B[i,j]



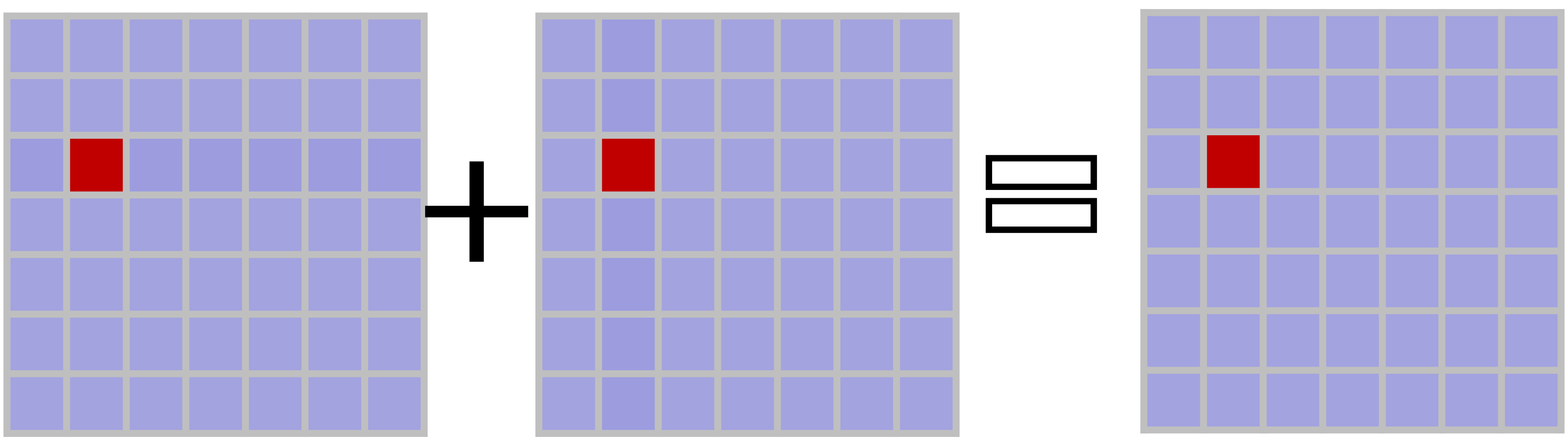
Matrix Addition

Work Analysis

– $T_1(n) = O(n^2)$

Critical Path Analysis

– $T_\infty(n) = O(n^2)$



Ex: Matrix Addition

```
plus(A,B,C,i,j)
  C[i,j] = A[i,j] + B[i,j];

pBadBadMatrixAdd(A, B, C, n)
  for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        spawn plus(A,B,C,i,j);
```

Ex: Matrix Addition

```
plus(A,B,C,i,j)
  C[i,j] = A[i,j] + B[i,j];

pBadBadMatrixAdd(A, B, C, n)
  for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        spawn plus(A,B,C,i,j);
```

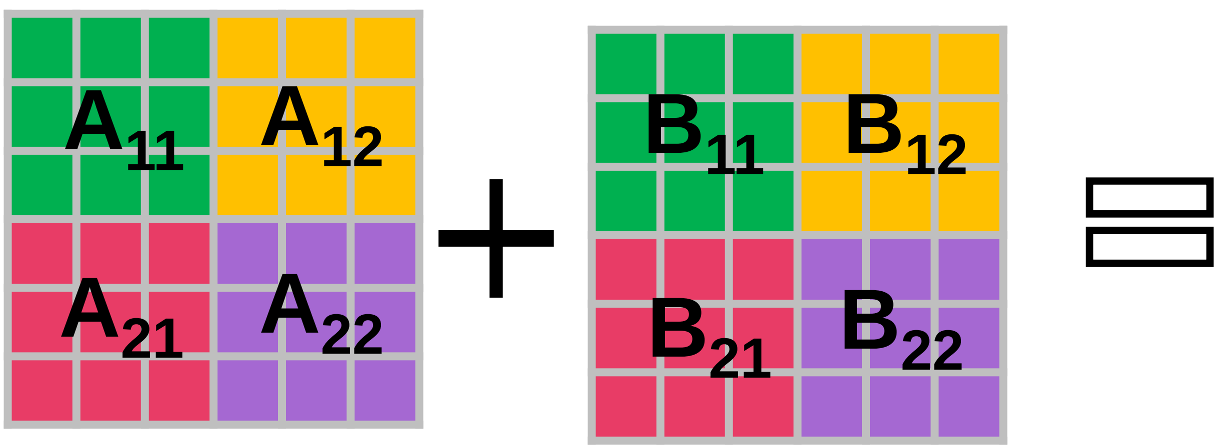
Loop construct:
Execute this line n^2 times!

Critical Path: $T_\infty(n) = O(n^2)$

Matrix Addition

Basic idea: divide-and-conquer

- Divide matrix into four quadrants.
- Sum each part separately.
- All four parts can happen in parallel!



Matrix Addition

```
pMatAdd(A, B, C, i, j, n)
  if (n == 1)
    C[i,j] = A[i,j]+B[i,j];
  else
    spawn pMatAdd(A,B,C,i,j,n/2);
    spawn pMatAdd(A,B,C,i,j+n/2,n/2);
    spawn pMatAdd(A,B,C,i+n/2,j,n/2);
    spawn pMatAdd(A,B,C,i+n/2,j+n/1,n/2);
    synch;
```

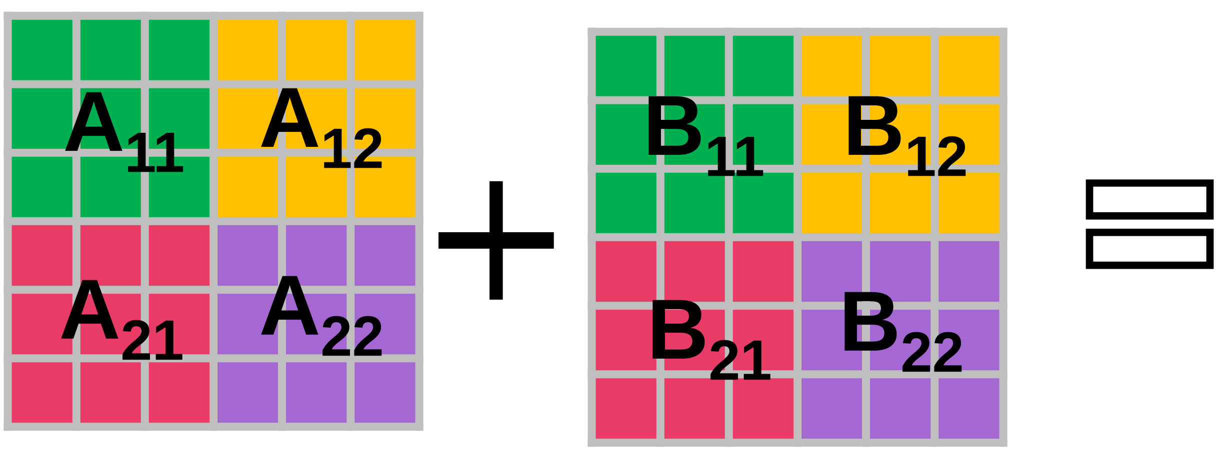
Matrix Addition

Work Analysis

– $T_1(n) = 4T_1(n/2) + O(1) = O(n^2)$

Critical Path Analysis

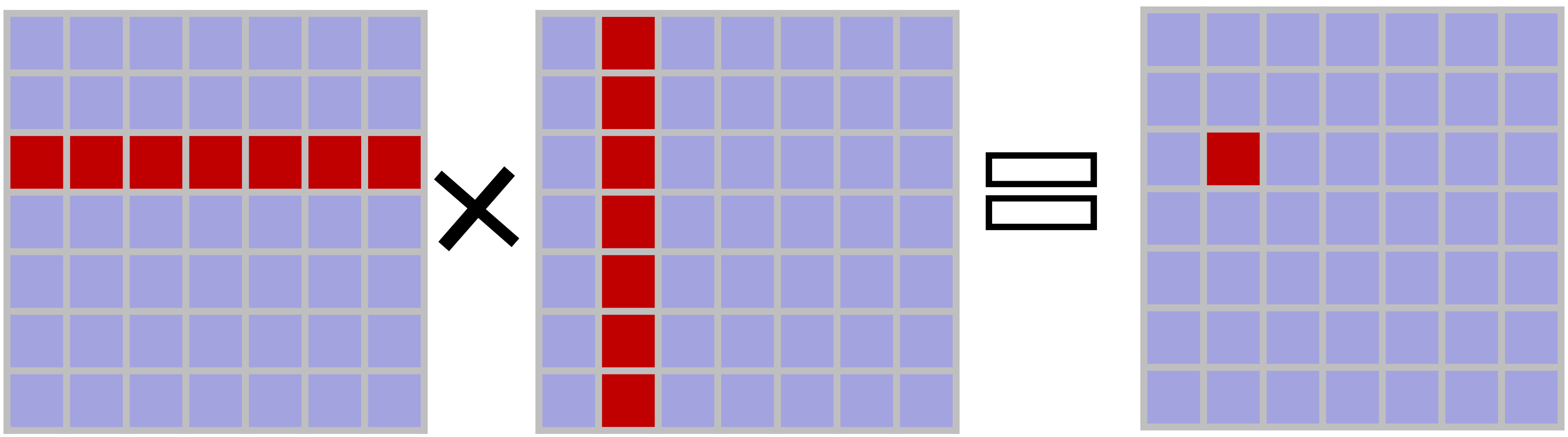
– $T_\infty(n) = T_\infty(n/2) + O(1) = O(\log n)$



Example: Matrix Multiplication

Given: two matrices A[n,n] and B[n,n]

Calculate: matrix C = AB

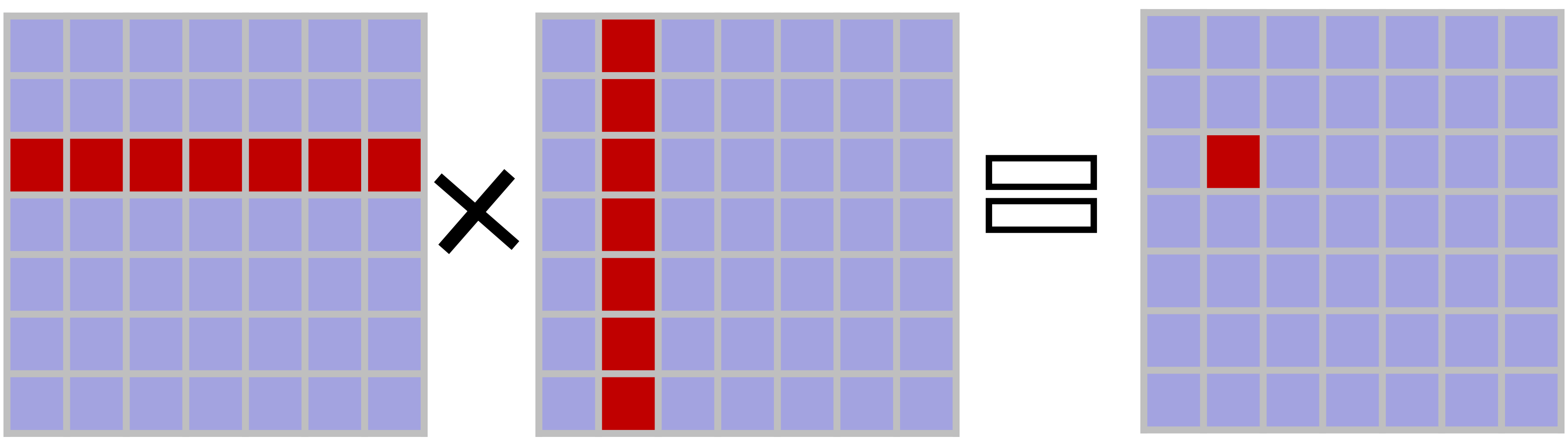


$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

Matrix Multiplication

Time: $O(n^3)$

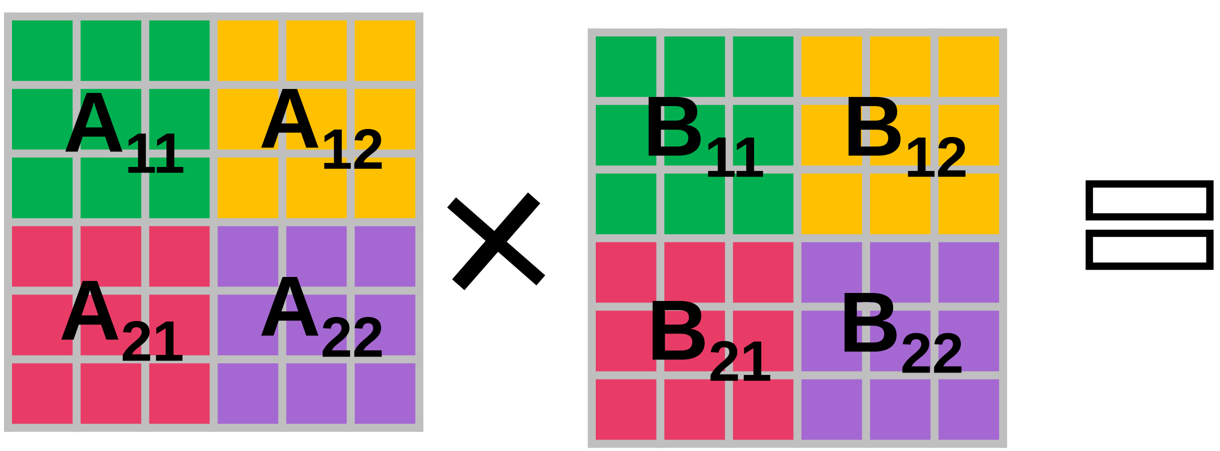
```
Multiply(A,B)
    for i = 1 to n do
        for j = 1 to n do
            C[i,j] = 0
            for k = 1 to n do C[i,j] += A[i,k] * B[k,j]
```



Matrix Multiplication

Divide-and-Conquer

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$



Ex: Matrix Addition

```
pMatMult(A, B, C, n)
  if (n == 1) C=A*B;
  else
    spawn pMatMult(A11, B11, C11, n/2);
    spawn pMatMult(A12, B21, T11, n/2);
    spawn pMatMult(A11, B12, C12, n/2);
    spawn pMatMult(A12, B22, T12, n/2);
    spawn pMatMult(A21, B12, C22, n/2);
    spawn pMatMult(A22, B22, T22, n/2);
    spawn pMatMult(A21, B11, C21, n/2);
    spawn pMatMult(A22, B21, T21, n/2);
    synch;
    spawn pMatAdd(C, T, C, n)
    synch;
```

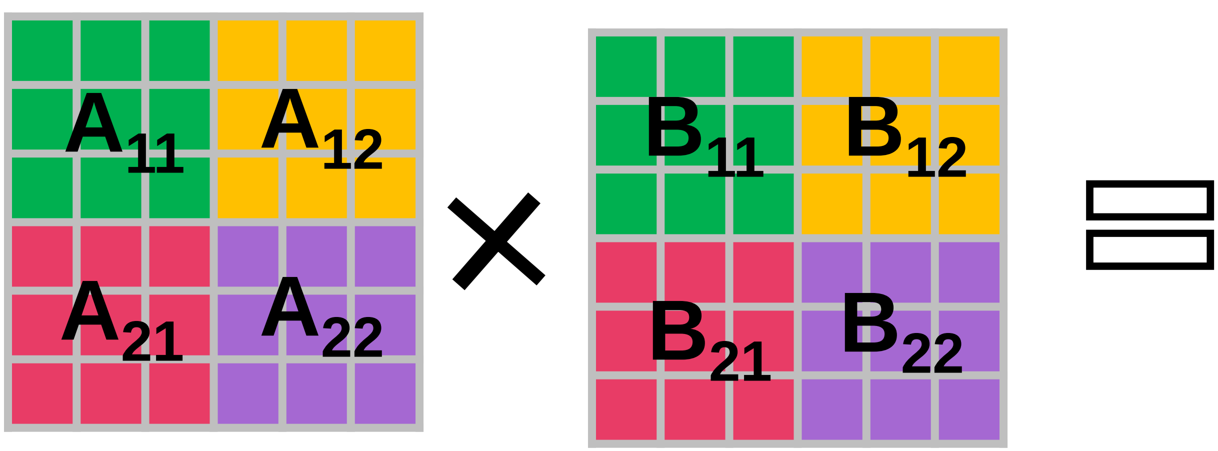
Matrix Multiplication

Work Analysis

- $T_1(n) = 8T_1(n/2) + 4O(n^2/4) = O(n^3)$

8 multiplications

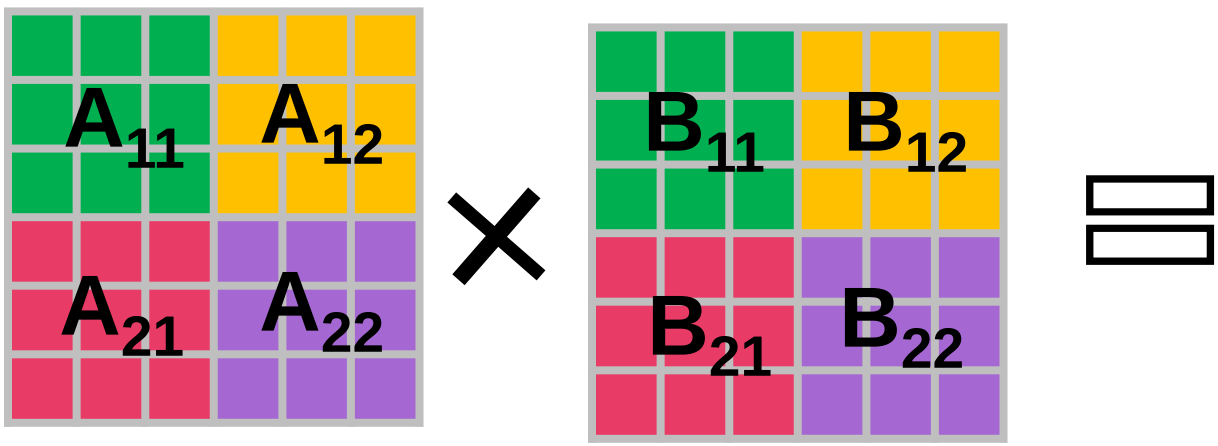
4 additions



Matrix Multiplication

Divide-and-Conquer

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$



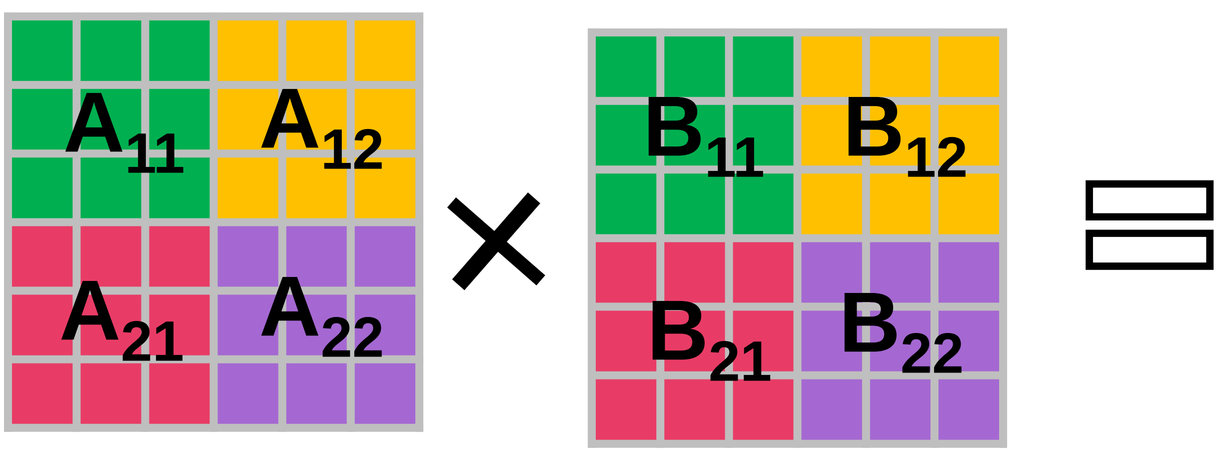
Matrix Multiplication

Critical Path Analysis

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

time for 1 multiplication:
all multiplications in parallel

time for one parallel addition:
all additions in parallel



Ex: Matrix Addition

```
pMatMult(A, B, C, n)
  if (n == 1) C=A*B;
  else
    spawn pMatMult(A11, B11, C11, n/2);
    spawn pMatMult(A12, B21, T11, n/2);
    spawn pMatMult(A11, B12, C12, n/2);
    spawn pMatMult(A12, B22, T12, n/2);
    spawn pMatMult(A21, B12, C22, n/2);
    spawn pMatMult(A22, B22, T22, n/2);
    spawn pMatMult(A21, B11, C21, n/2);
    spawn pMatMult(A22, B21, T21, n/2);
    synch;
    spawn pMatAdd(C, T, C, n)
    synch;
```

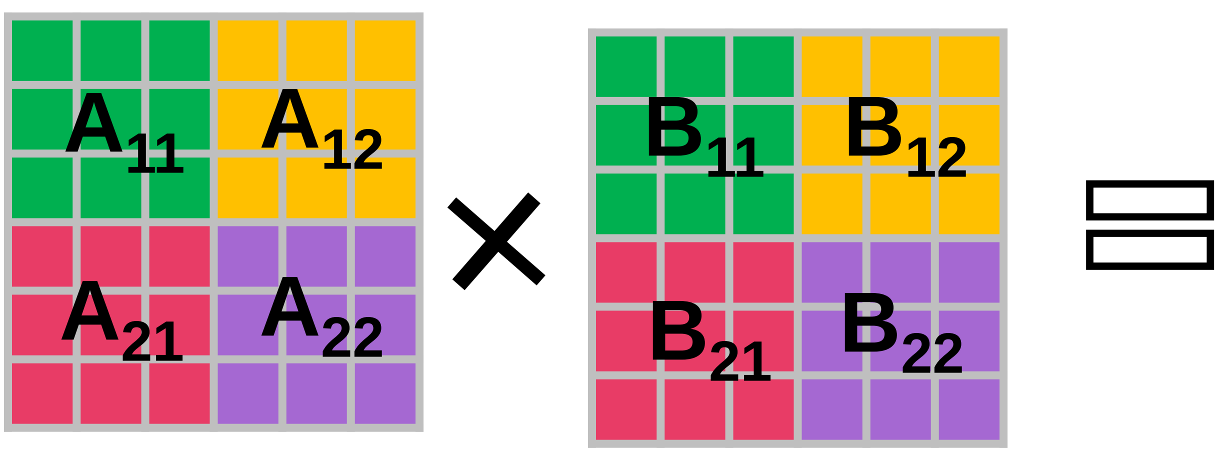
Matrix Multiplication

Work Analysis

– $T_1(n) = 8T_1(n/2) + O(4) = O(n^3)$

Critical Path Analysis

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



Matrix Multiplication

Alternate version:

- Uses no temporary space.
- But has a longer critical path.

Trade-off: time vs. space

Exercise: come up with a version that uses no extra space!

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 = spawn pMergeSort(A[1..n/2], n/2)
    Y = spawn pMergeSort(A[n/2+1, n], n/2)
    synch;
    A = Merge(X, Y);
```

Parallel Sorting

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

Critical Path Analysis

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

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:

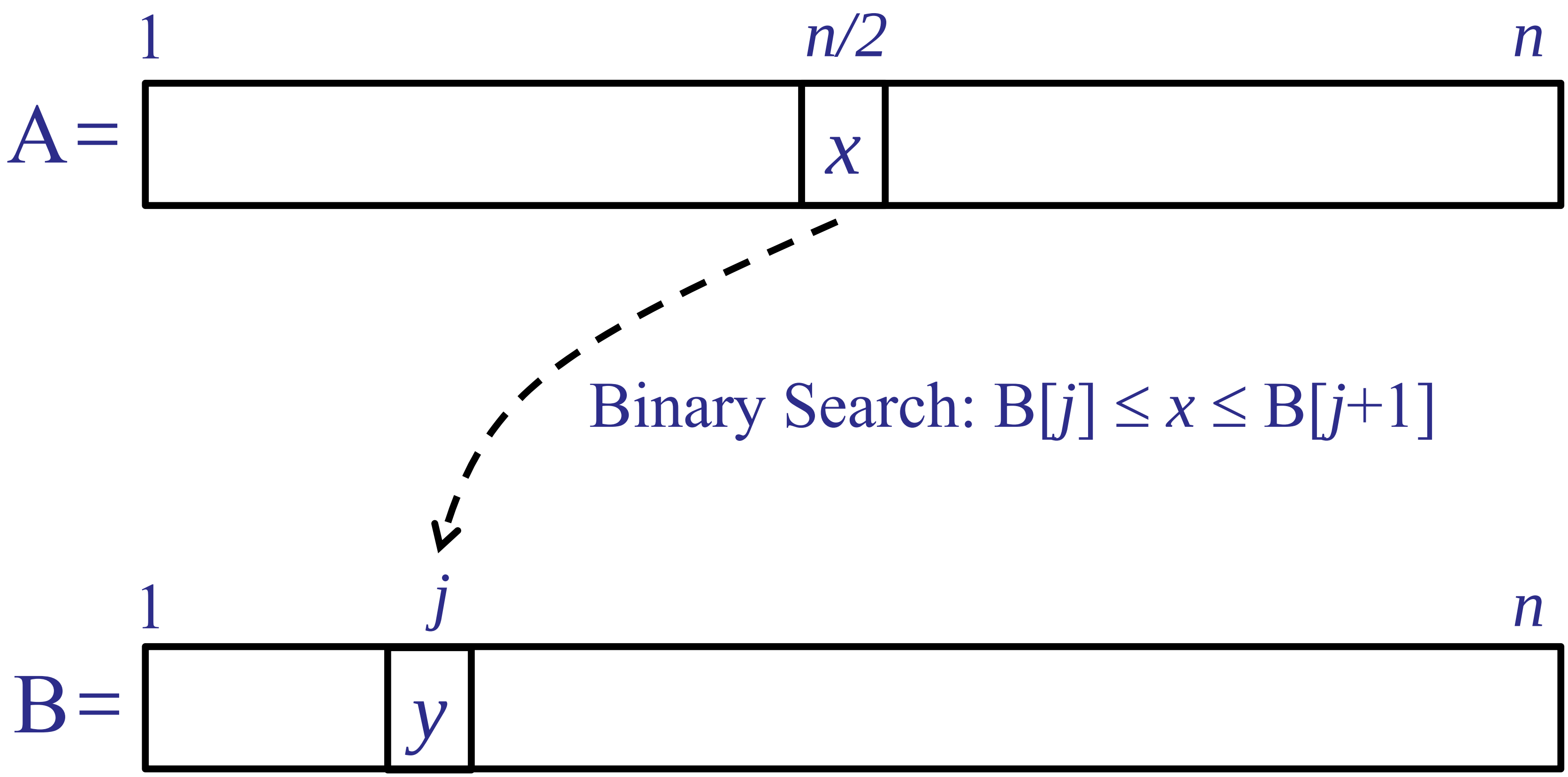
```
X = spawn Merge(A[1..n/2], B[1..n/2])
Y = spawn Merge(A[n/2+1..n], B[n/2+1..n])
```

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

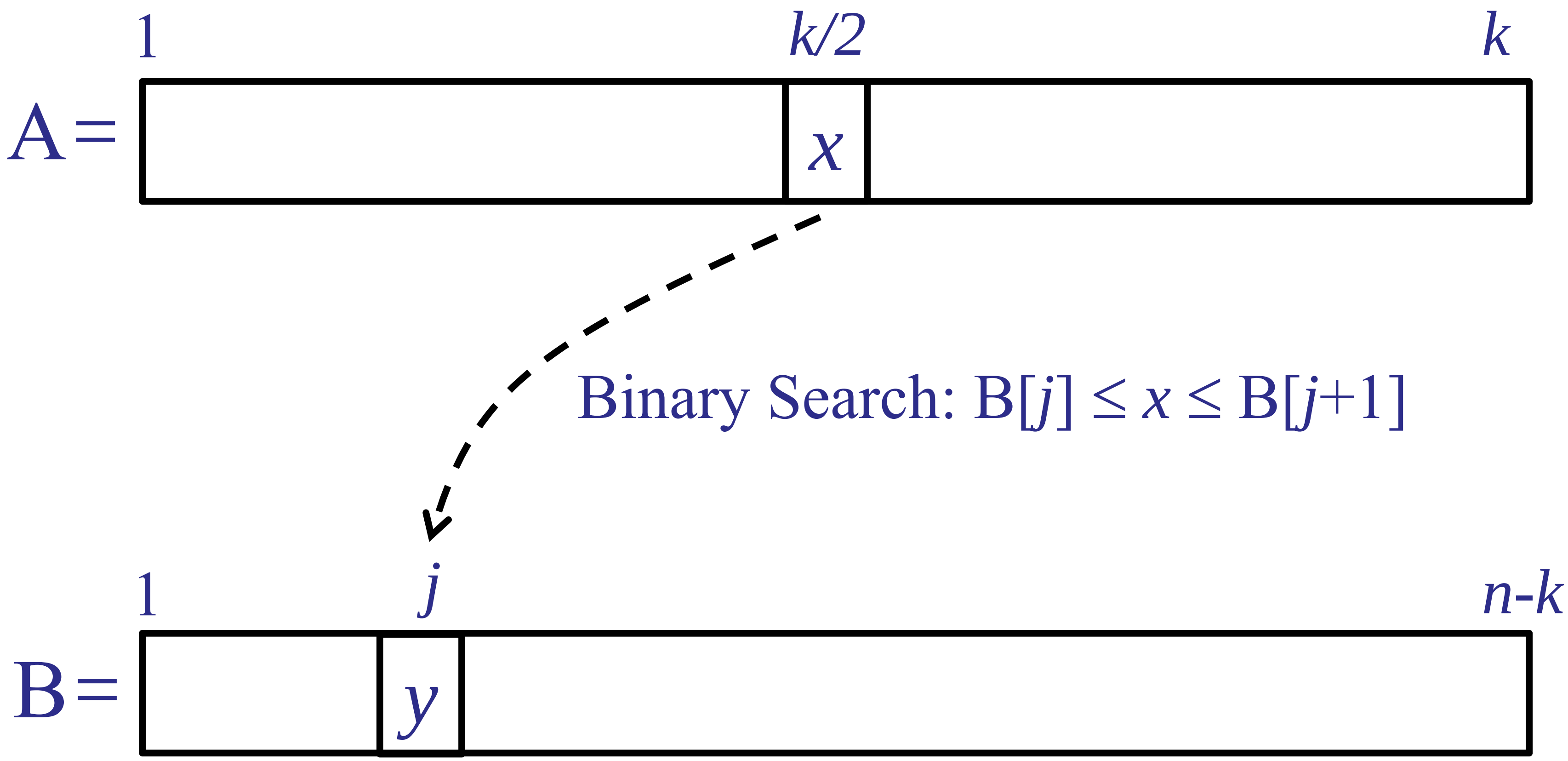


Recurse: **pMerge**(A[1..n/2], B[1..j])
pMerge(A[n/2+1..n], B[j+1..n])

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]
    spawn pMerge(A[1..k/2],B[1..j],C[1..k/2+j])
    spawn pMerge(A[k/2+1..l],B[j+1..m],C[k/2+j+1..n])
    synch;
```

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 work done by 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: $k/2 + (n - k) = n - (k/2) < n - (n/4) < 3n/4$
- $T_\infty(n) \le T_\infty(3n/4) + O(\log n)$

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

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: $\frac{1}{4} \leq \alpha \leq \frac{3}{4}$
- $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + O(\log n)$

$$\approx 2T_1(n/2) + O(\log n)$$

$$= O(n)$$

Parallel Sorting

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

Critical Path Analysis

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

So far today…

- Model for parallel algorithms
  - Dynamic multithreading
- Metrics for analyzing parallel programs
  - Work
  - Critical path
- Two parallel algorithms
  - Matrix multiplication
  - MergeSort

Parallel Programming is Hard

Parallel Problems

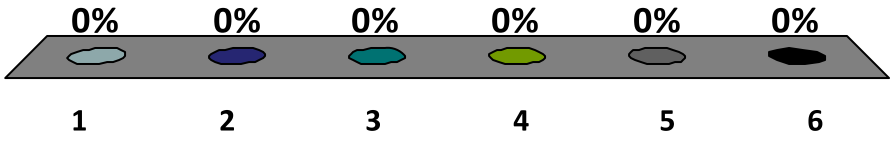
```
class Counter{
  int cnt = 0;

  void pInc(){
    cnt = cnt + 1;
  }
}

Counter c = new counter();
spawn c.pInc();
spawn c.pInc();
sync();
```

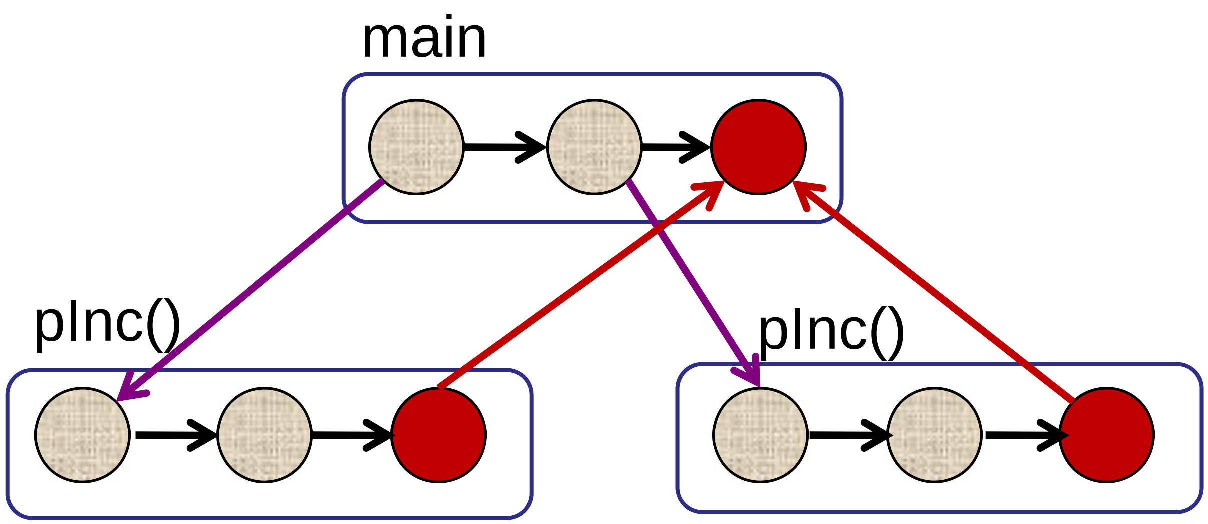
What is the final value of the counter?

1. 0
2. 1
3. 2
4. 1 or 2
5. 0 or 1
6. 0 or 1 or 2



Parallel Problems

Computation DAG:



Parallel Problems

Processor 1

- x = cnt.read() = 0
- cnt = x+1;

Processor 2

- y = cnt.read() = 1
- cnt = y+1

Result: cnt = 2;

Parallel Problems

Processor 1

- x = cnt.read() = 0
- cnt = x+1;

Processor 2

- y = cnt.read() = 0
- cnt = y+1

Result: cnt = 1;

Parallel Problems

Race condition:

- Different parallel interleavings lead to different outcomes.
- Design algorithms to avoid race conditions!

When does a race condition occur:

- Two threads can access the same shared memory.
- Those two threads can happen in either order.

Parallel Problems

Another example:

```
class Milk{
        quantity = 2 liters;

        checkMilk():
                if (quantity < 1 liter) then
                        buy 1 liter milk;
        }
}
```

Beware: if used in parallel, too much milk!

Avoiding Race Conditions

Locks:

- A lock ensures that only one process can access a *critical section* at a given time.

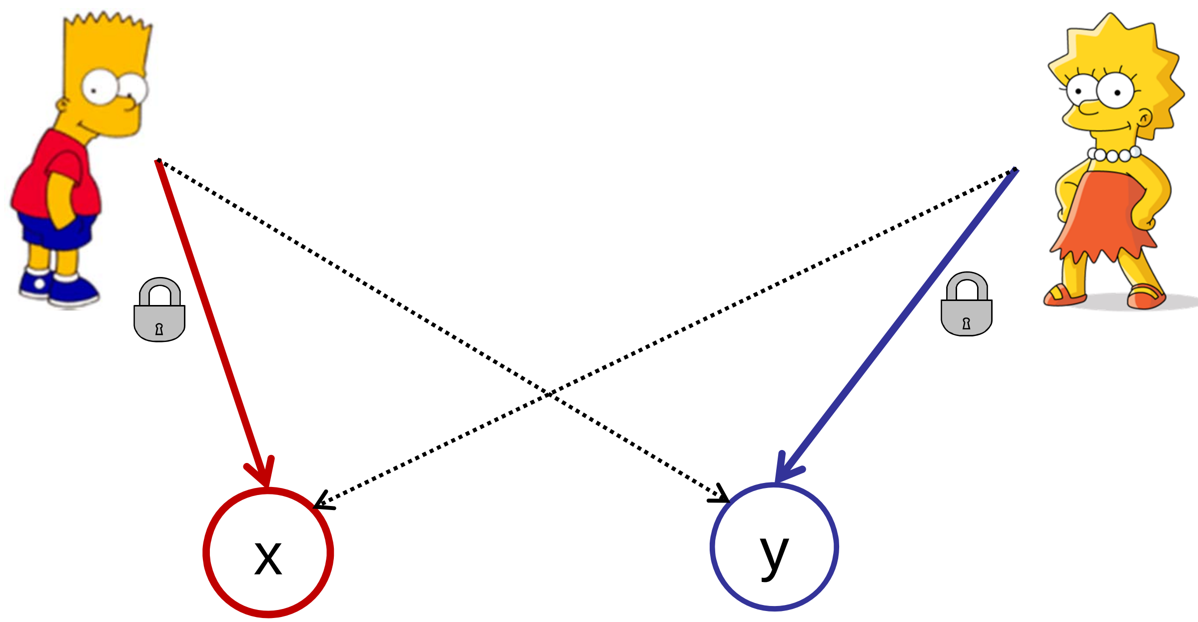
```
checkMilk():
     lock(milk);
     if (quantity < 1 liter) then
          buy 1 liter milk;
     unlock(milk);
}
```

- Only one process can access milk at a time!

Parallel Problems

Deadlocks (see: Dining Philosophers)

- Process A: lock(x)
- Process B: lock(y)
- Process A: lock(y)
- Process B: lock(x)



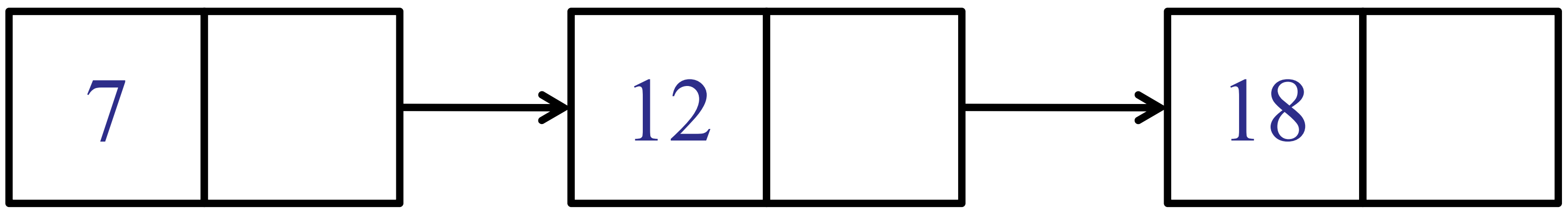
Linked List

- Support:
  - insert
  - delete
  - search

- Parallelism:
  - Many operations will run at the same time.
  - E.g., a database with many simultaneous users.

Linked List (sorted)

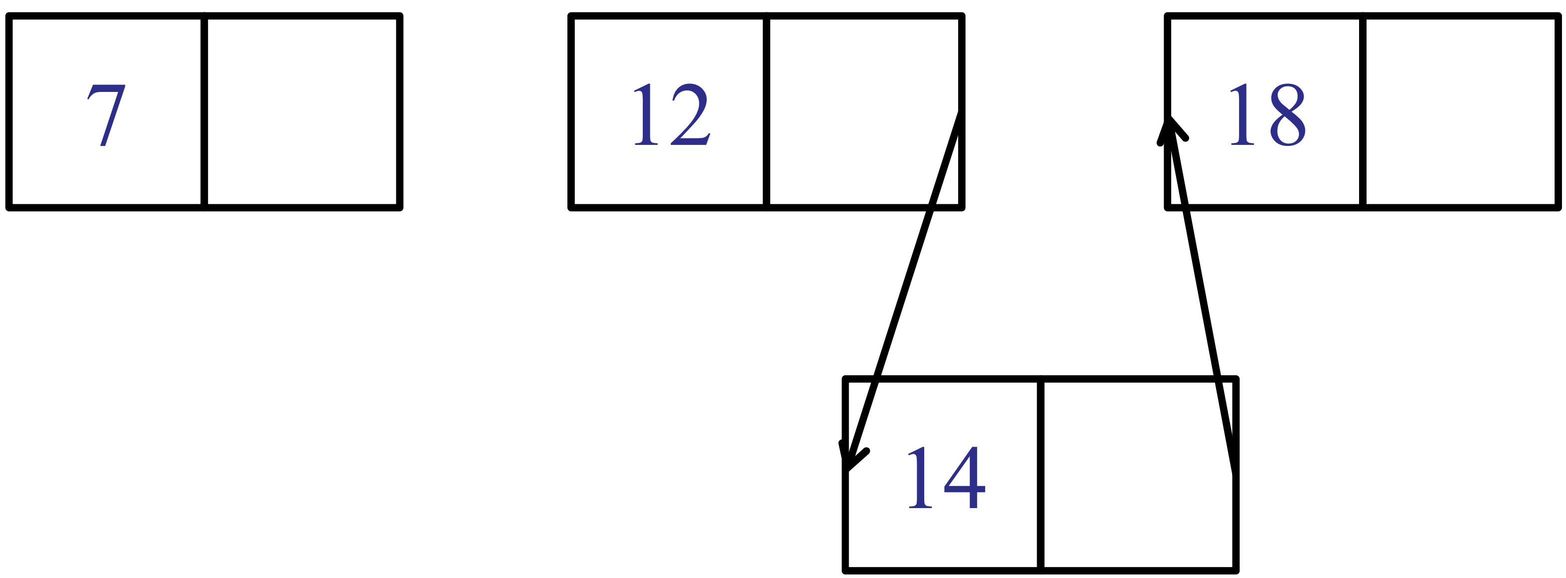
Problem 1: Concurrent insert/delete



insert(14) → after 12

Linked List (sorted)

Problem 1: Concurrent insert/delete

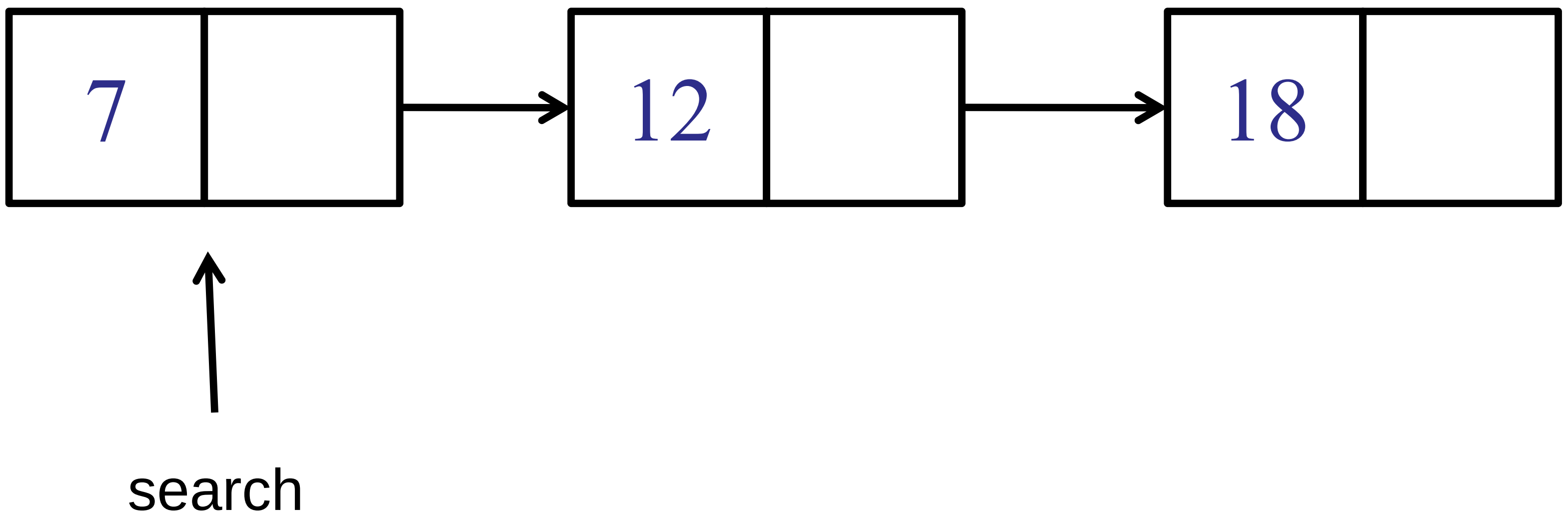


insert(14) → after 12

delete(12)

Linked List (sorted)

Problem 2: infinite search

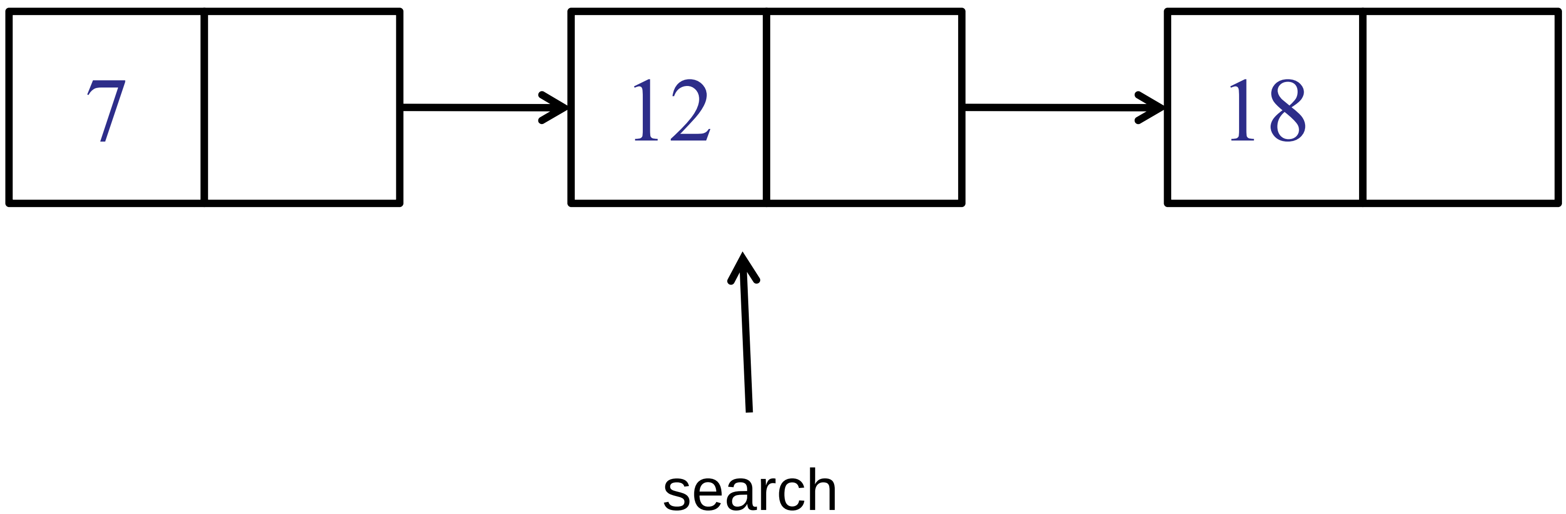


insert(12.1), insert(12.11), insert(12.111), …

search(18)

Linked List (sorted)

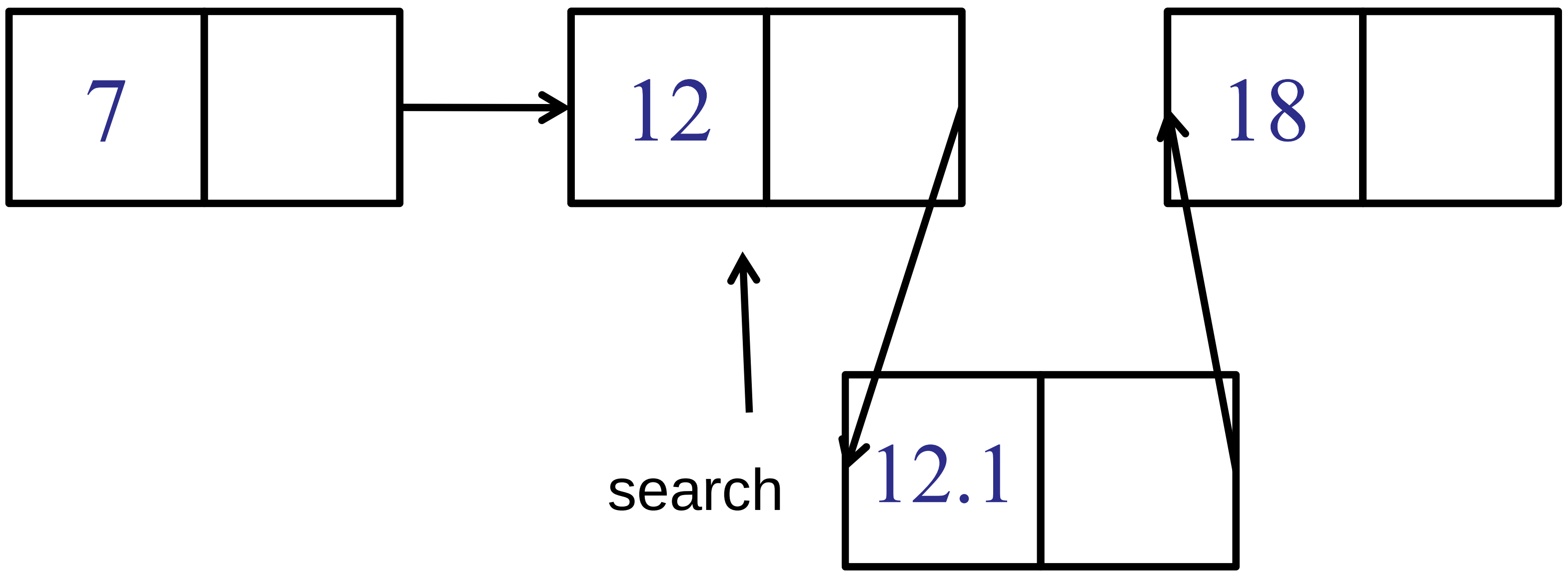
Problem 2: infinite search



insert(12.1), insert(12.11), insert(12.111), …
search(18)

Linked List (sorted)

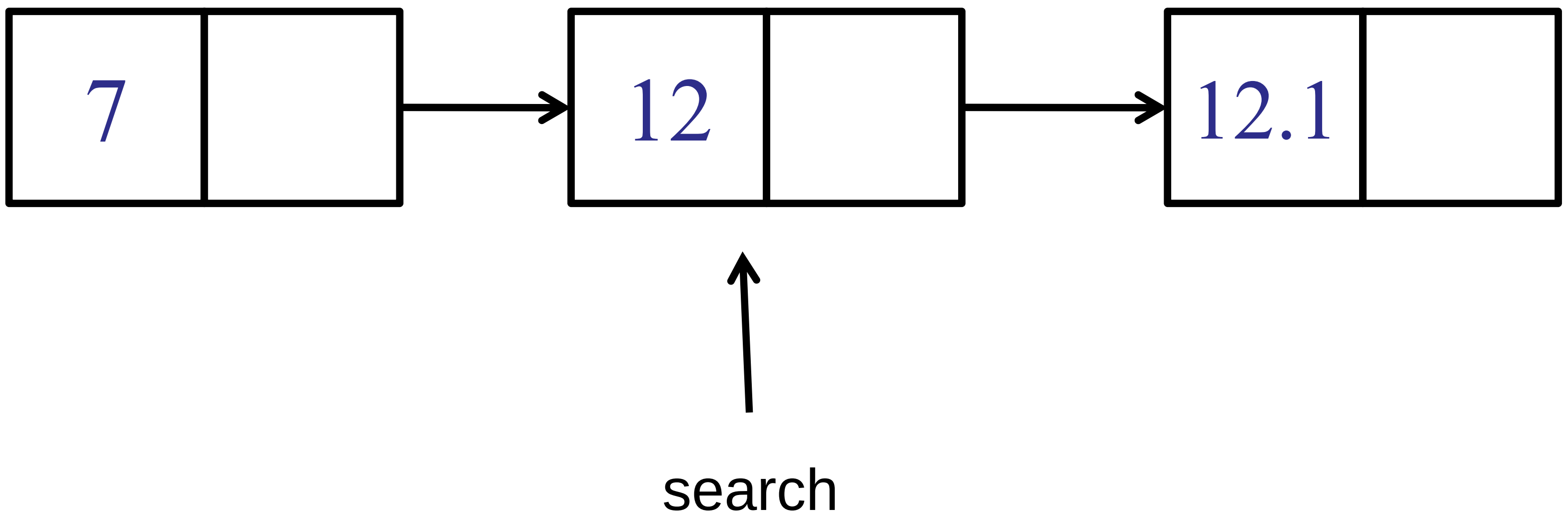
Problem 2: infinite search



insert(12.1), insert(12.11), insert(12.111), …
search(18)

Linked List (sorted)

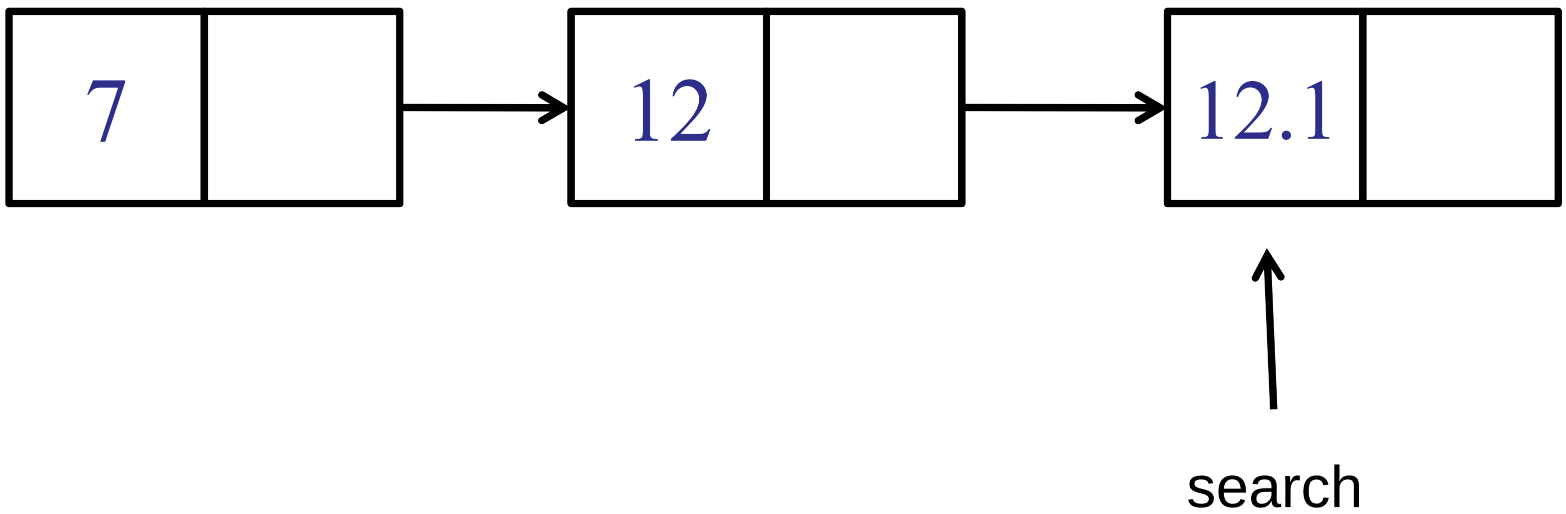
Problem 2: infinite search



insert(12.1), insert(12.11), insert(12.111), …
search(18)

Linked List (sorted)

Problem 2: infinite search

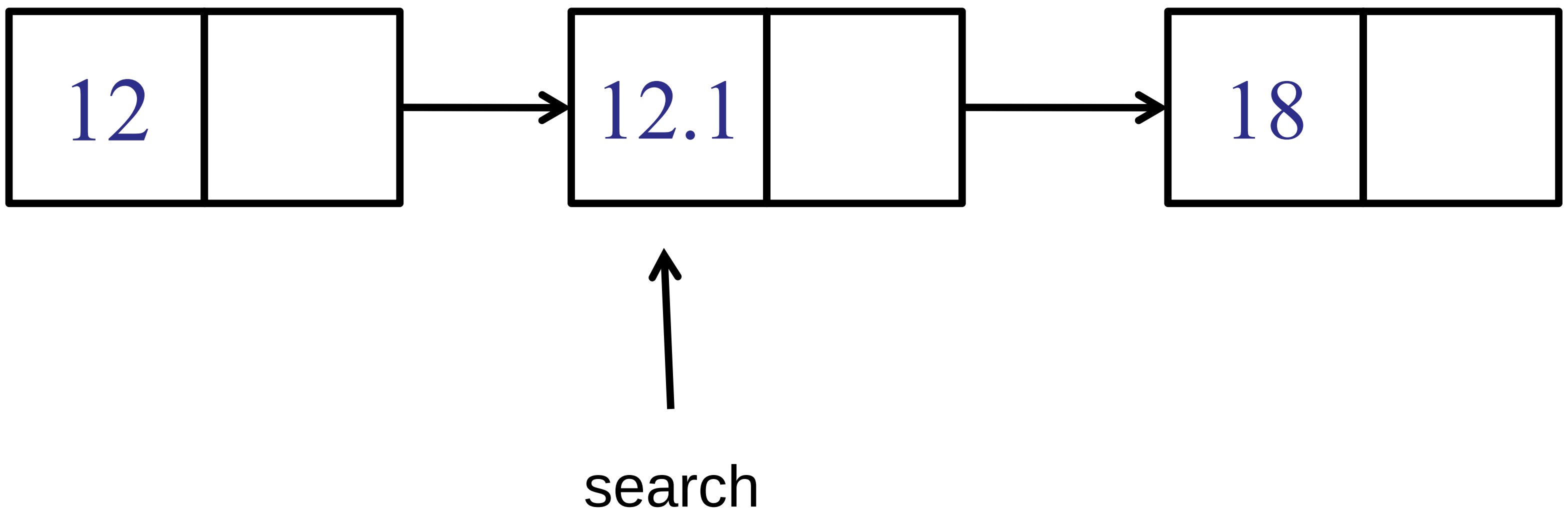


insert(12.1), insert(12.11), insert(12.111), …

search(18)

Linked List (sorted)

Problem 2: infinite search

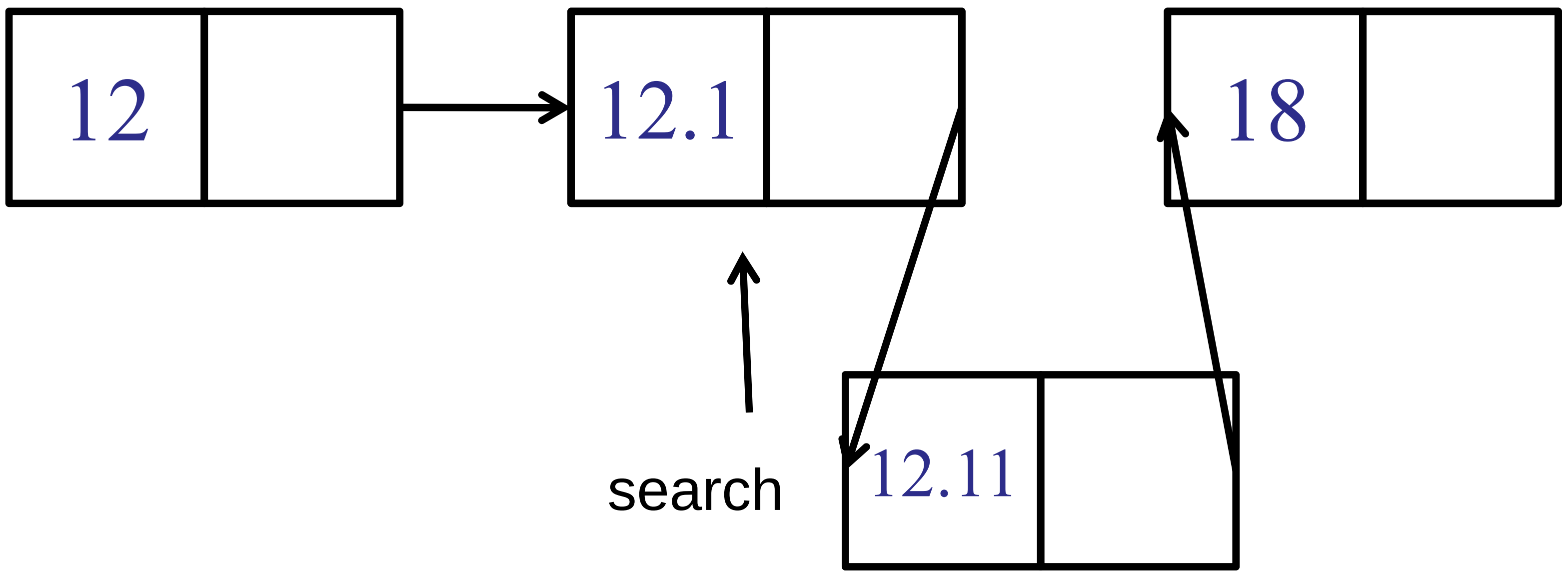


insert(12.1), insert(12.11), insert(12.111), …

search(18)

Linked List (sorted)

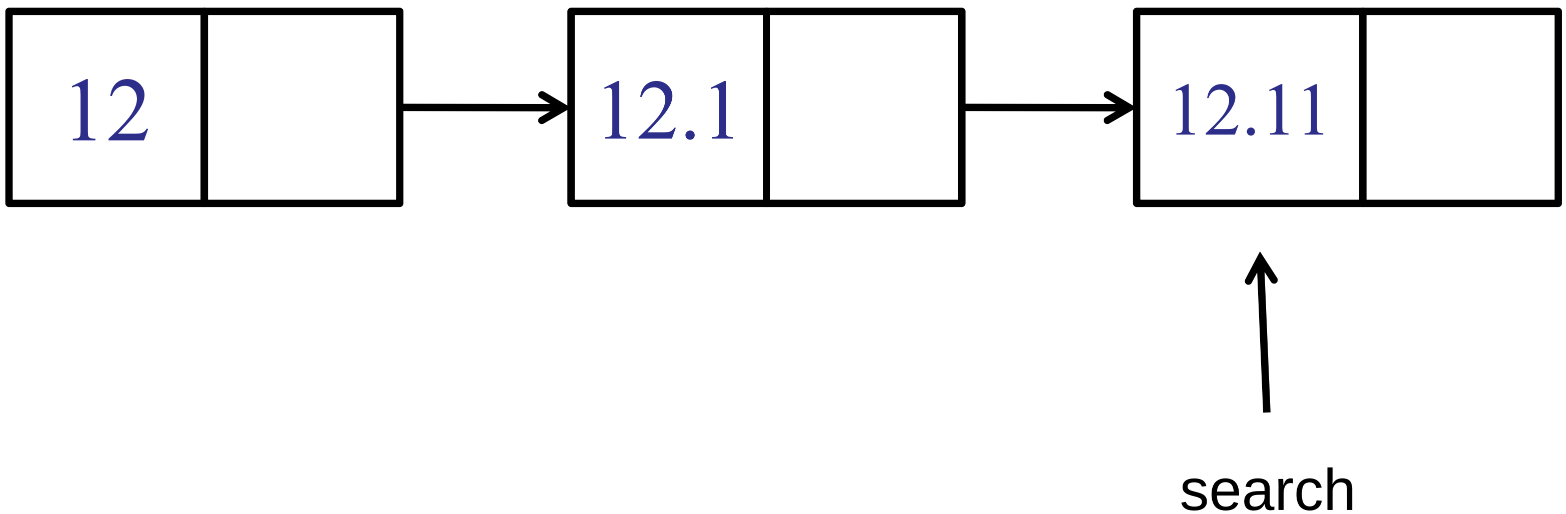
Problem 2: infinite search



insert(12.1), insert(12.11), insert(12.111), …
search(18)

Linked List (sorted)

Problem 2: infinite search



insert(12.1), insert(12.11), insert(12.111), …
search(18)

Linked List

- Support: only integers
  - insert
  - ~~delete~~
  - search

- Parallelism:
  - Many operations will run at the same time.
  - E.g., a database with many simultaneous users.

Linked List

```
insert(int x){

    LLnode node = new LLnode(x);

    lock(this);

    node.setNext(this.next);
    this.setNext(node);

    unlock(this);
}
```

NB: Fake Java syntax for locks…

Concurrent Search Trees

AVL Trees

- Option 1: Lock entire tree tree insert.
  - Poor concurrency.
  - Only one process can insert at a time.
  - Can processes search during an insert?

Concurrent Search Trees

AVL Trees

- Option 1: Lock entire tree tree insert.
- Option 2: Hand-over-hand locking
  - Let x = insertion point.
  - Lock x
  - Repeat:
    - Lock parent of x.
    - Rotate (if necessary).
    - Unlock x.
    - x = x.parent

MultiThreaded Java

- Basic class: java.lang.thread
  - Extend the thread class to create a thread.
  - Override: public void run(){ … }

- Basic interface: Runnable
  - Implement Runnable to create a thread
  - public void run(){ … }

MultiThreaded Java

```
class DBSearch extends Thread{
  static Database dbMain;
  static found = false;
  int searchKey;

  DBSearch(int i){searchKey = i;}

  public void run(){
     dbMain.search(searchKey);
  }

  public static void main(…){
     DBSearch[] dbS[1000]
     for (int i=0; i<1000; i++){
           dbS[i] = new DBSearch(i);
           dbS[i].start();
     }
  }
```

MultiThreaded Java

Synchronization:

– Ensure that only one process accesses shared data at a time by declaring an object *synchronized*.

```
public synchronized void doSomething(){
   dbMain.search(searchKey);
}
```

MultiThreaded Java

Example:

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }
    public synchronized int value() { return c; }
}
```

See: Oracle Java Concurrency Tutorial

Dynamic Multithreading in Java

Fork/Join executors

- Maintains a pool of threads (ForkJoinPool)
- Each task implements ForkJoinTask
- A task either does computation, or spawns other (smaller) ForkJoinTasks.
- An ExecutorService distributed tasks to worker threads in the thread pool via a work-stealing algorithm.

Today

- Model for parallel algorithms
  - Dynamic multithreading
- Metrics for analyzing parallel programs
  - Work and Critical path
- Two parallel algorithms
  - Matrix multiplication and MergeSort
- Problems in Parallelism
  - Race conditions and deadlock
- MultiThreaded Java

Quick Wrap Up

"If you need your software to run twice as fast, hire better programmers.

But if you need your software to run more than twice as fast, use a better **algorithm.**"

-- *Software Lead at Microsoft*

Algorithms

Object-oriented programming

Java

Goals for the Semester

Algorithms:

- Design of efficient algorithms
- Analysis of algorithms

Implementation:

- Solve real problems
- Analyze and profile performance
- Improve performance via better algorithms

Quick Wrap Up

Goals of CS2020:

- Learn some Java.
- Learn some algorithms.
- Solve lots of problems.

Quick Wrap Up

Goals of CS2020:

- Learn some Java.
- Learn some algorithms.
- Solve lots of problems.

I hope everyone has had fun…

Quick Wrap Up

Goals of CS2020:

- Learn some Java.
- Learn some algorithms.
- Solve lots of problems.

I hope everyone has had fun…

If you ever want to chat about algorithms, just drop by my office…

Quick Wrap Up

Goals of CS2020:

- Learn some Java.
- Learn some algorithms.
- Solve lots of problems.

I hope everyone has had fun…

If you ever want to chat about algorithms, just drop by my office…

And it's over… congratulations!