departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Parallel Performance

Concurrency and Parallelism — 2019-20

Master in Computer Science

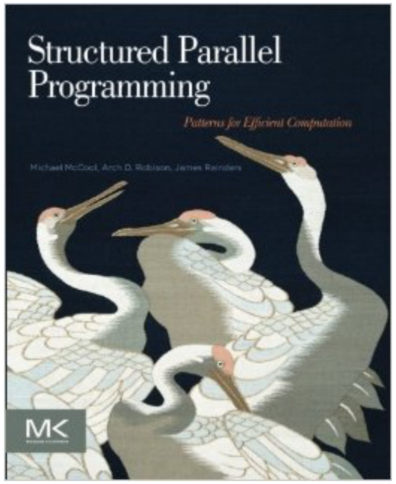
(Mestrado Integrado em Eng. Informática)

Joao Lourenço <joao.lourenco@fct.unl.pt>

Outline

- Performance scalability
  - Work-span model
  - Brent's lemma

  - Bibliography:
    - **Chapter 2** of book
      McCool M., Arch M., Reinders J.;
      Structured Parallel Programming: Patterns for Efficient Computation;
      Morgan Kaufmann (2012);
      ISBN: 978-0-12-415993-8



Amdhal's Law

If 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on!

But…

- Can all the applications be decomposed into just a serial part and a parallel part? For my application, what speedup should I expect?

- Most applications are not embarrassing parallel, they have a dependencies between code blocks and have a complex organization

Cilk+ fib() implementation

```
int fib(int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
   cilk_sync;
    return x+y;
  }
}
```

Launch thread

This is a "future"

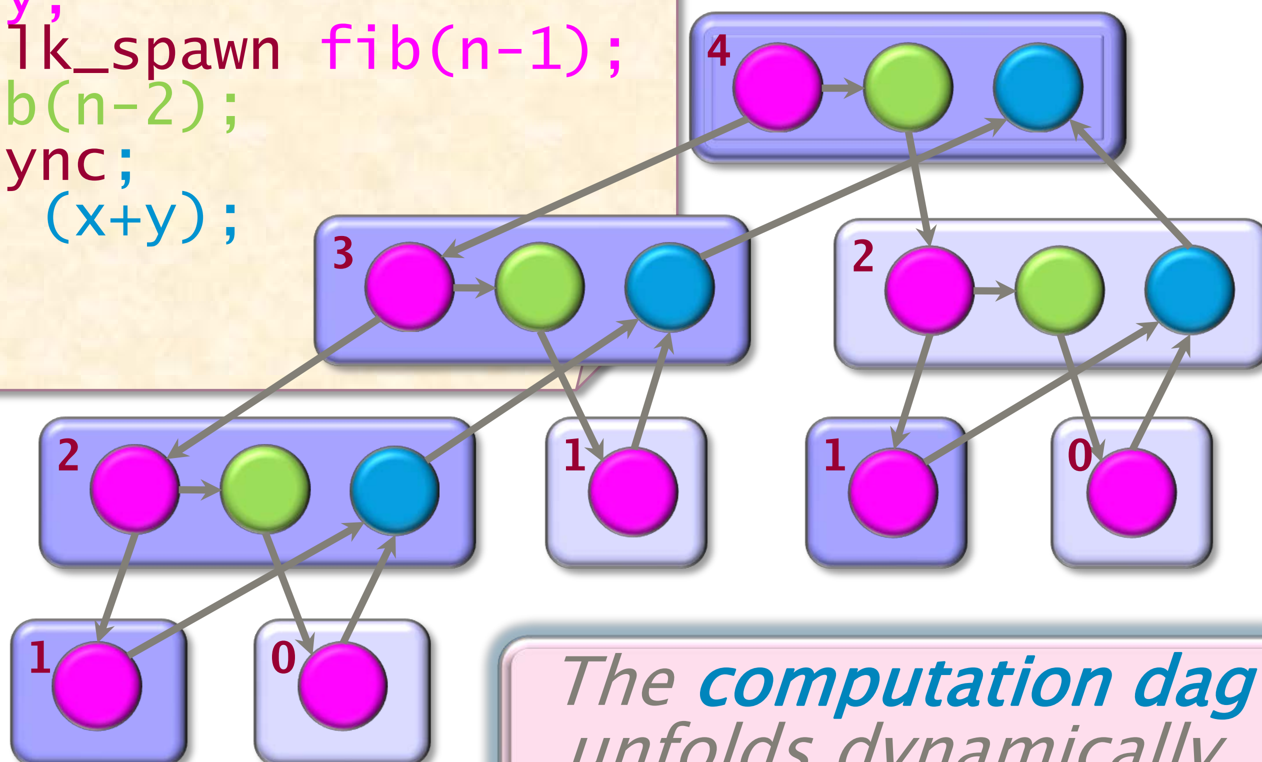
Main thread

Wait for "future"

Execution model

```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x+y);
  }
}
```

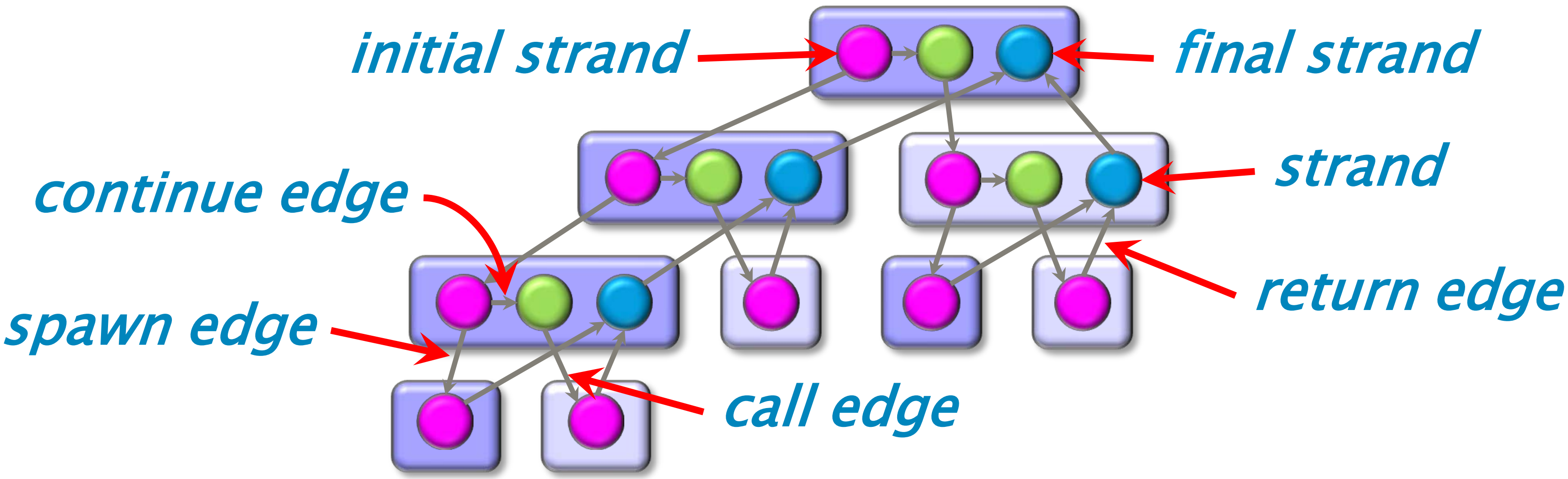
Example:
`fib(4)`



"Processor oblivious"

*The **computation dag** unfolds dynamically.*

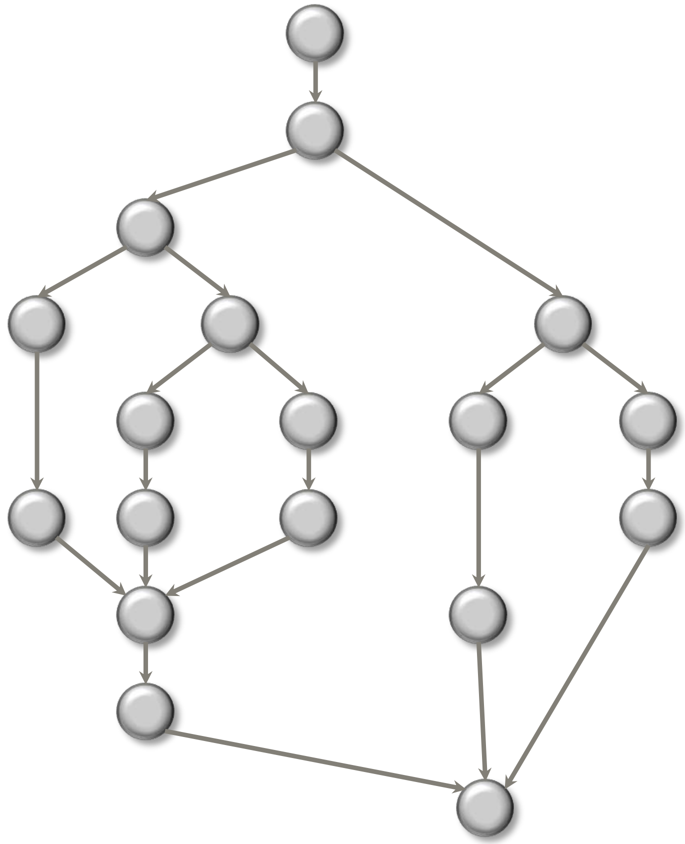
Computation DAG



- A parallel instruction stream is a dag $G = (V, E)$.
- Each vertex $v \in V$ is a strand : a sequence of instructions not containing a call, spawn, sync, or return (or thrown exception).
- An edge $e \in E$ is a spawn, call, return, or continue edge.
- Loop parallelism (cilk_for) is converted to spawns and syncs using recursive divide-and-conquer.

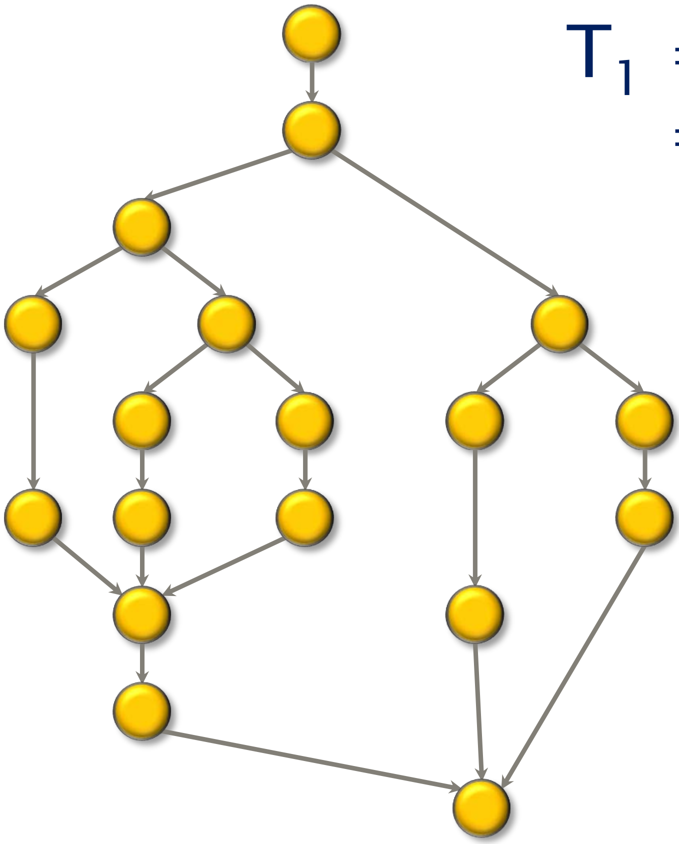
Performance Measures

T_P = execution time on P processors



Performance Measures

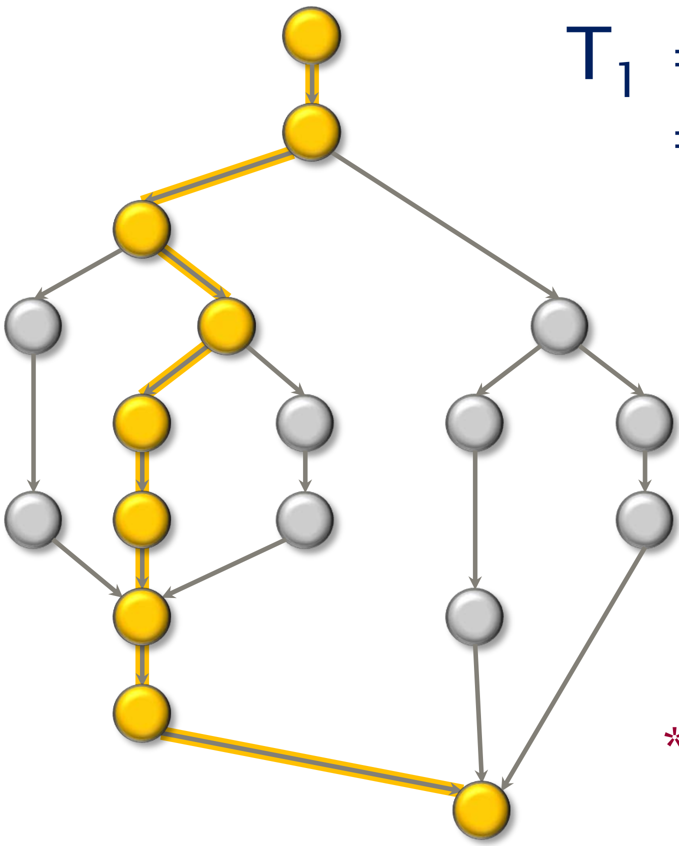
T_P = execution time on P processors



T_1 = *work*
= 18

Performance Measures

T_P = execution time on P processors



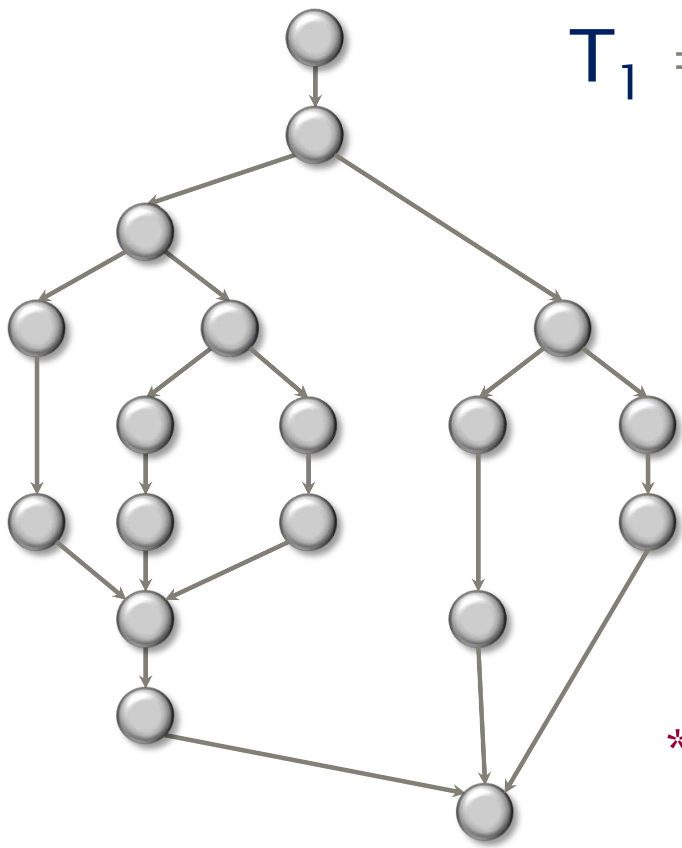
T_1 = *work* = 18

T_∞ = *span** = 9

*Also called *critical-path length* or *computational depth*.

Performance Measures

T_P = execution time on P processors



T_1 = ***work*** T_∞ = ***span****

WORK LAW

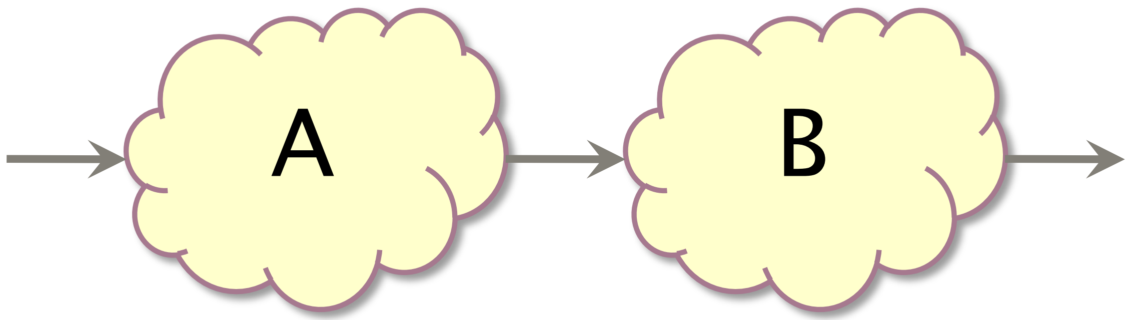
- $T_P \geq T_1 / P$

SPAN LAW

- $T_P \geq T_\infty$

*Also called *critical-path length* or *computational depth*.

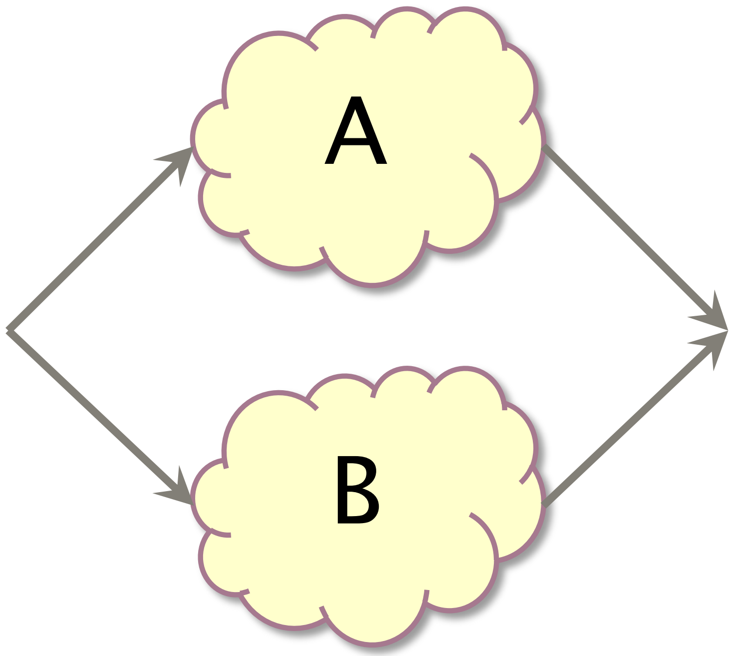
Serial Composition



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

Parallel Composition



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\}$

Speedup

Def. T_1/T_P = ***speedup*** on P processors.

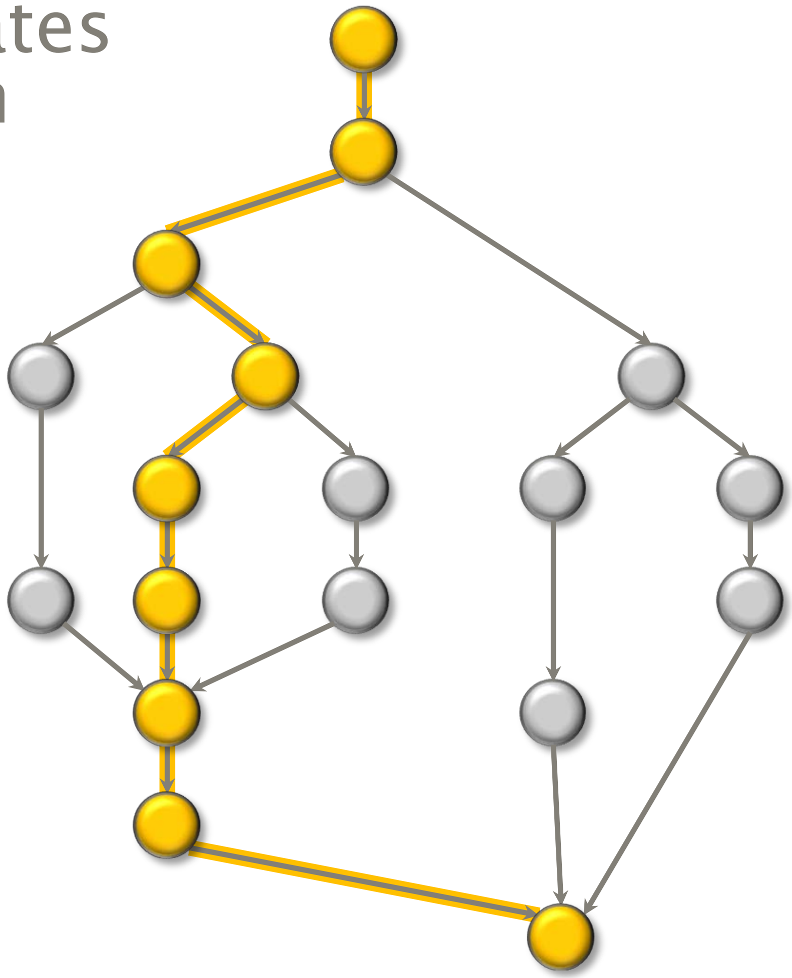
If $T_1/T_P = P$, we have ***(perfect) linear speedup***.

If $T_1/T_P > P$, we have ***superlinear speedup***, which is not possible in this performance model, because of the Work Law $T_P \geq T_1/P$.

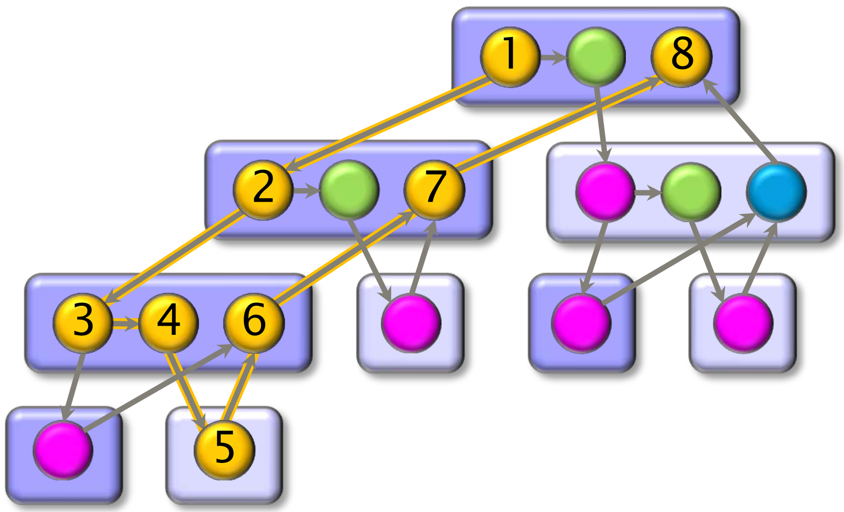
Parallelism

Because the Span Law dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = *parallelism*
= the average amount of work per step along the span.
= 18/9
= 2 .



Example: fib(4)



Assume for simplicity that each strand in `fib(4)` takes unit time to execute.

Work: $T_1 = 17$

Span: $T_\infty = 8$

Parallelism: $T_1/T_\infty = 2.125$

Using many more than 2 processors can yield only marginal performance gains.

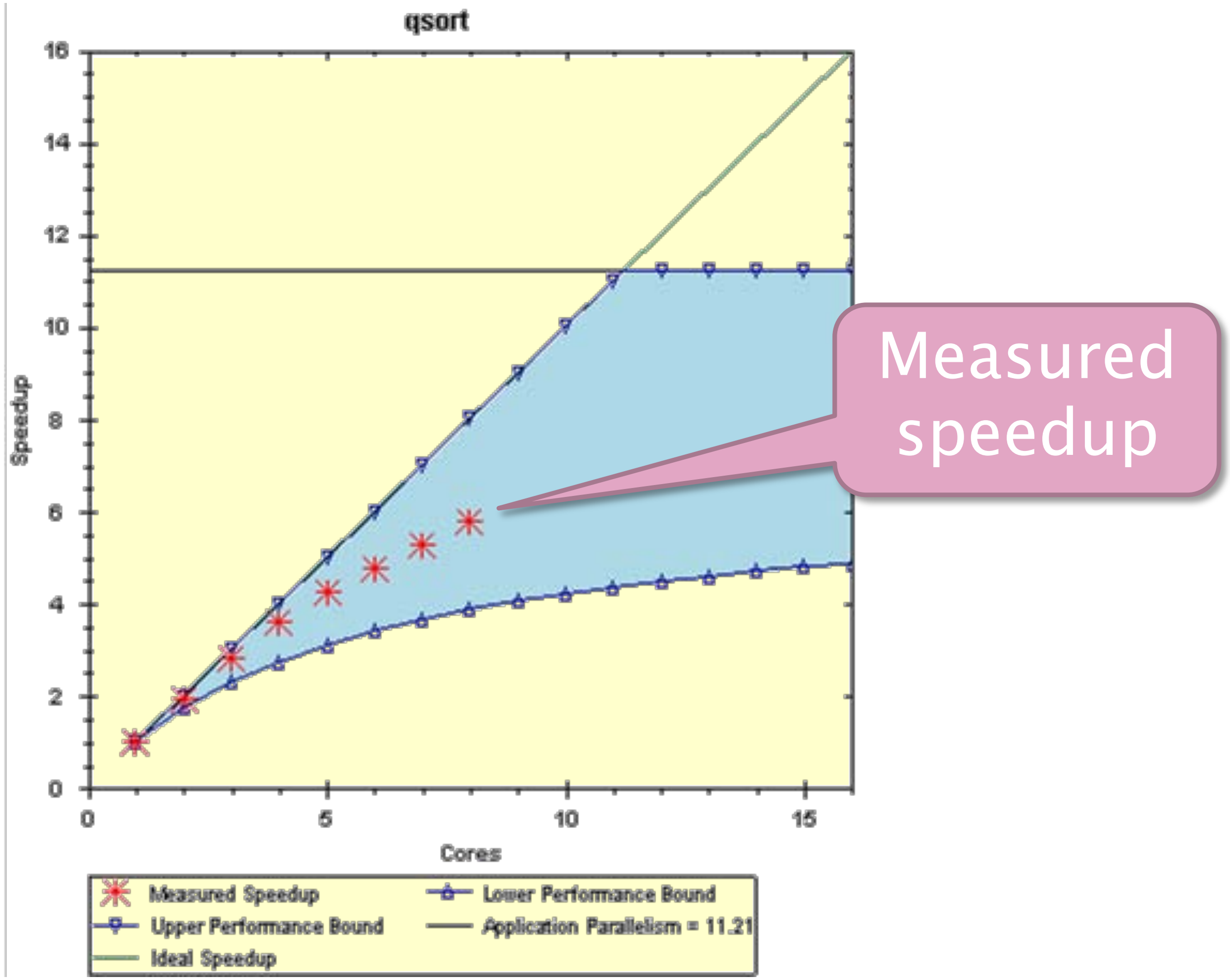
Quicksort Analysis

Note: the pointer arithmetic is invalid in this example, but I hope you get the idea!

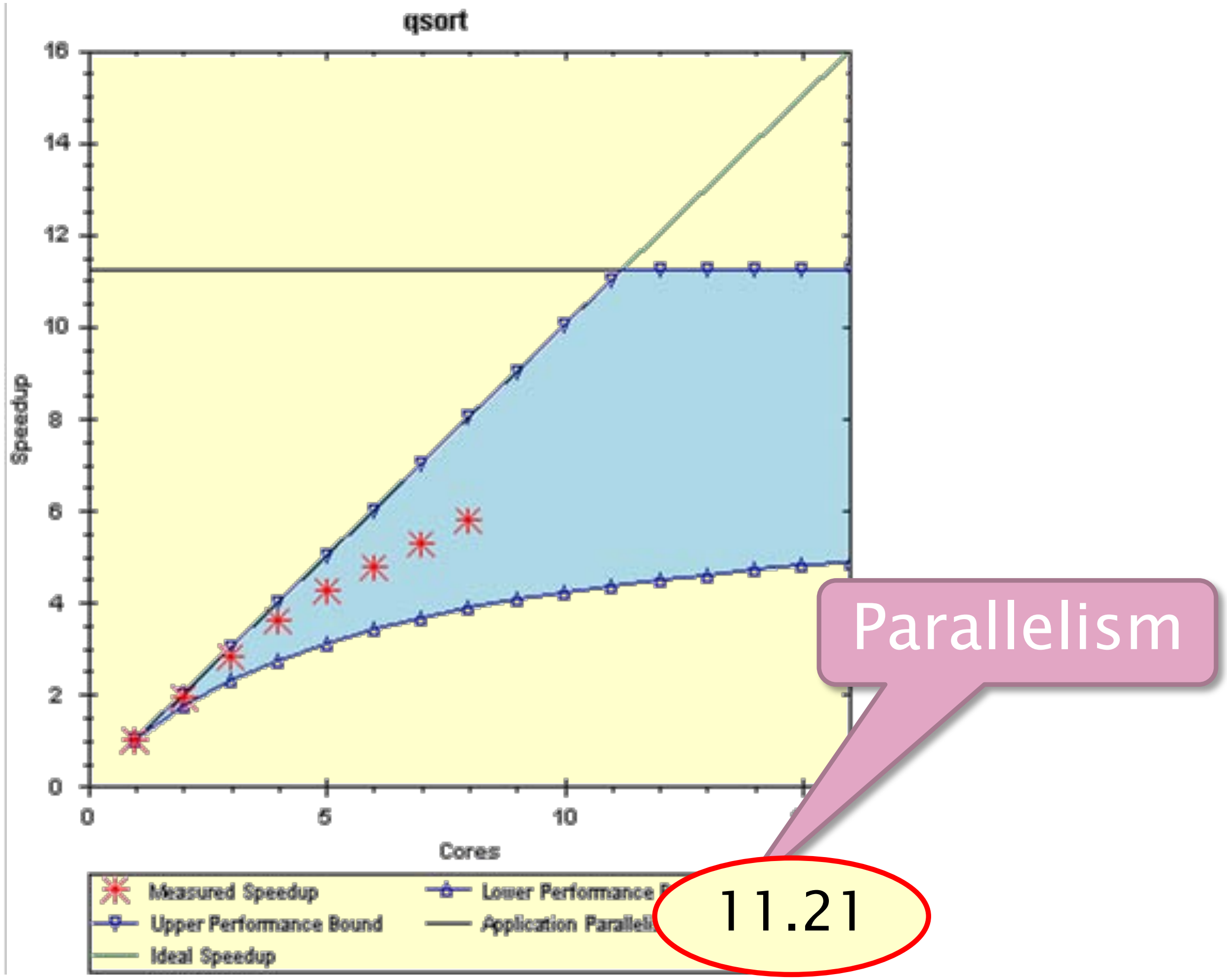
```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *))
{
  int p = partition(base, nel, width, compar);
  cilk_spawn qsort(&base[0], p, width, compar);
  qsort (&base[p+1], nel-(p+1), width, compar);
  cilk_sync;
}
```

Let’s analyze the sorting of 100,000,000 numbers!

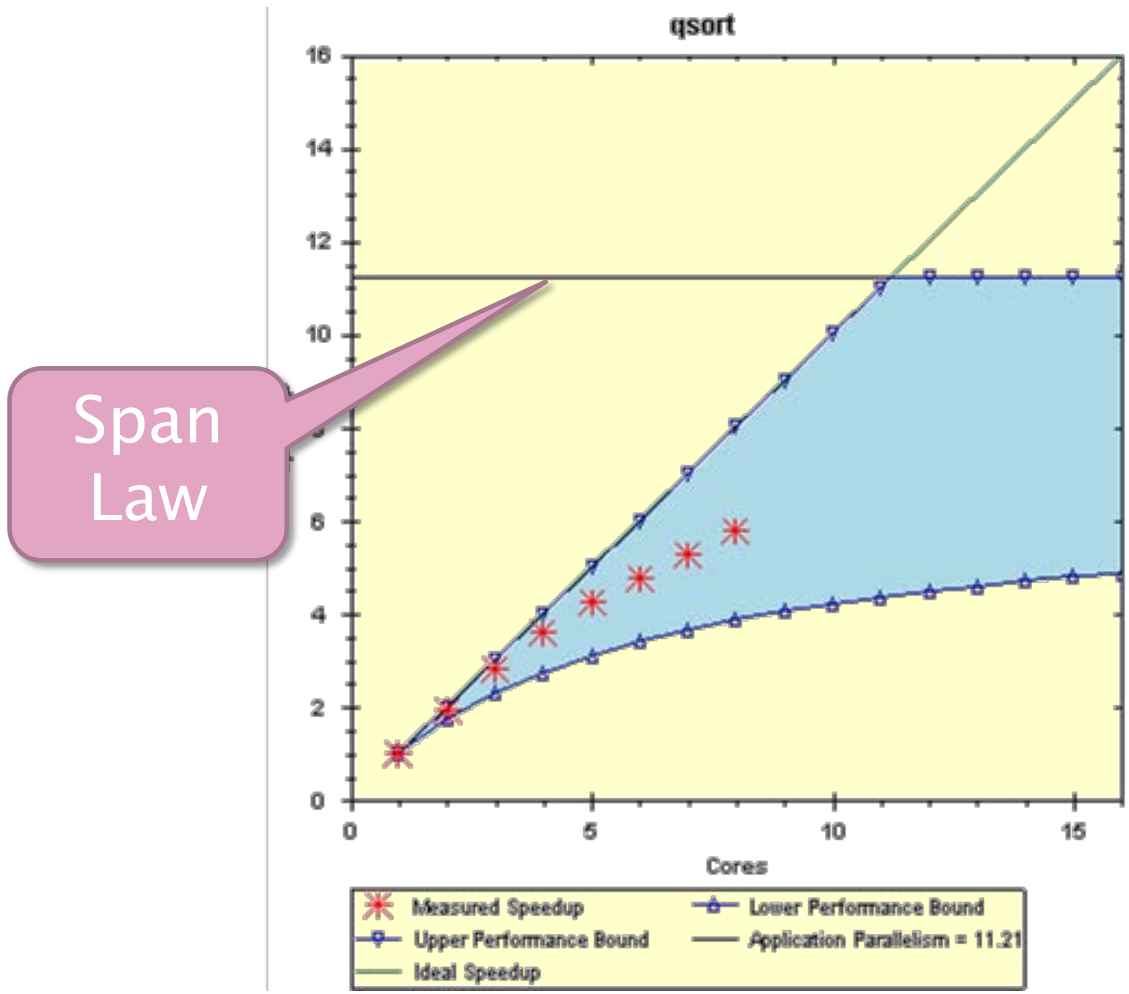
Parallel performance



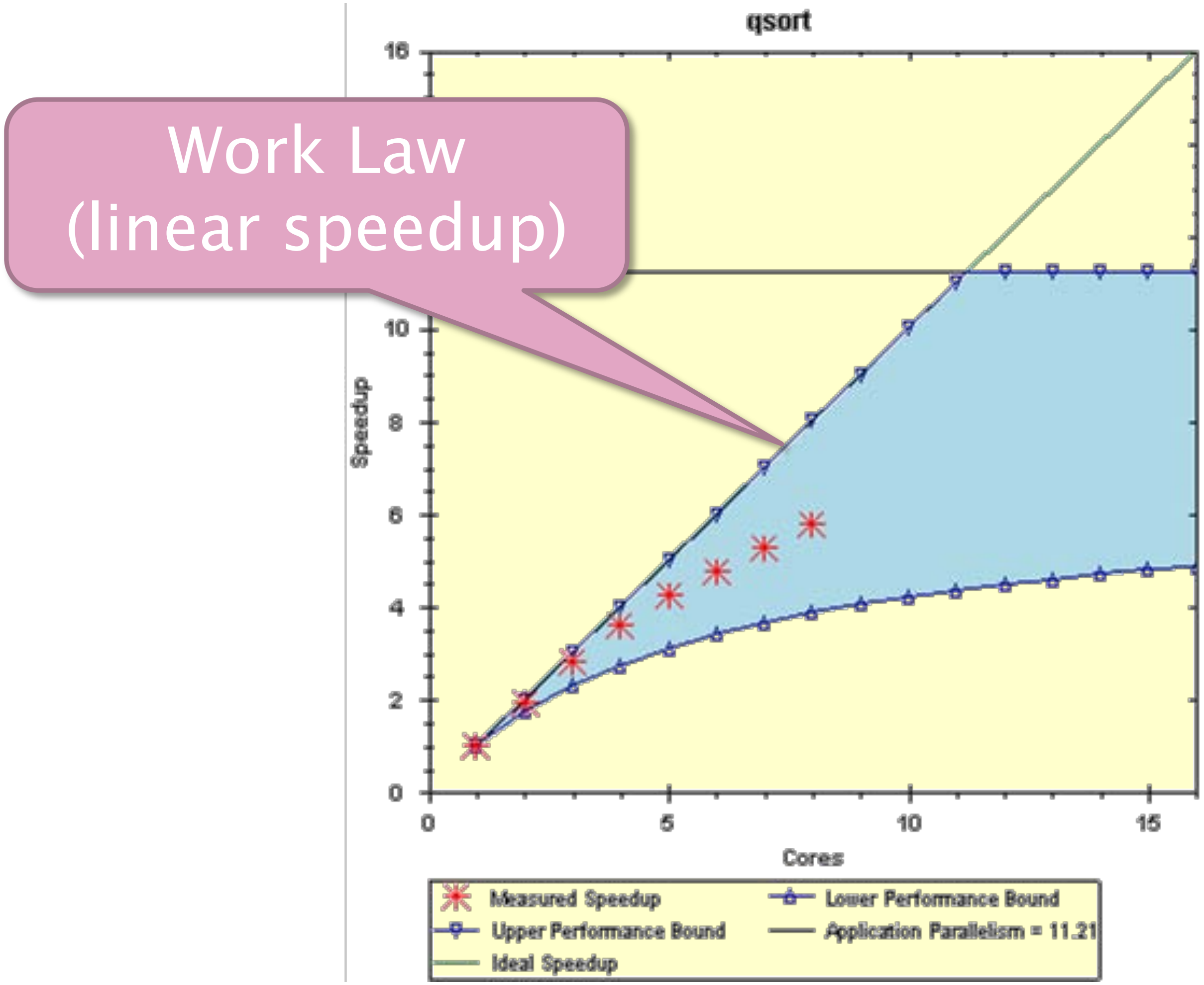
Parallel performance



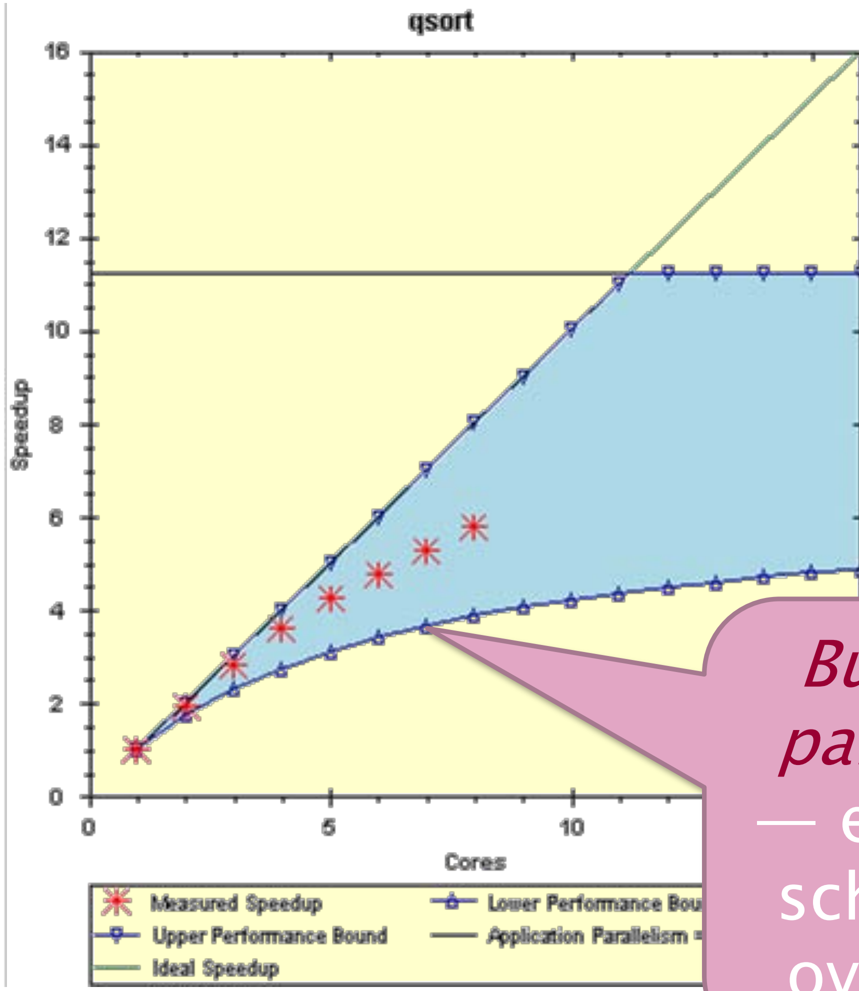
Parallel performance



Parallel performance



Parallel performance



Burdened parallelism — estimates scheduling overheads

Quicksort Analysis

Note: the pointer arithmetic is invalid in this example, but I hope you get the idea!

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *))

{
  int p = partition(base, nel, width, compar);

  cilk_spawn qsort(&base[0], p, width, compar);

  qsort (&base[p+1], nel-(p+1), width, compar);

  cilk_sync;

}
```

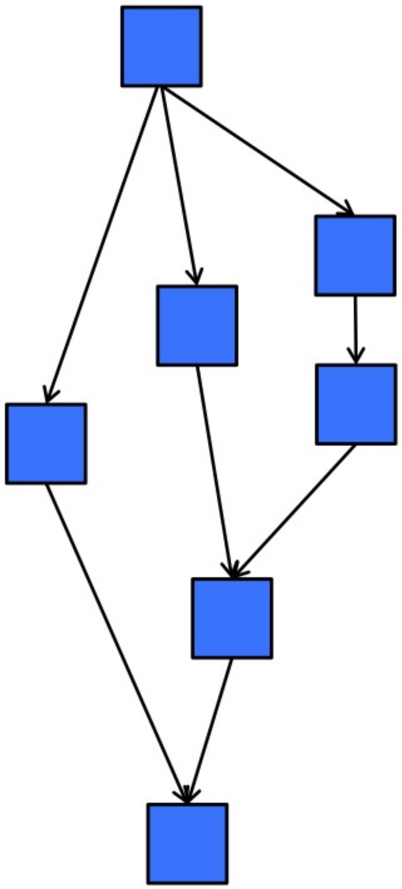
Expected Work = O(n lg n)
Expected Span = O(n)
=> Parallelism = O(lg n)

Interesting Practical Algorithms

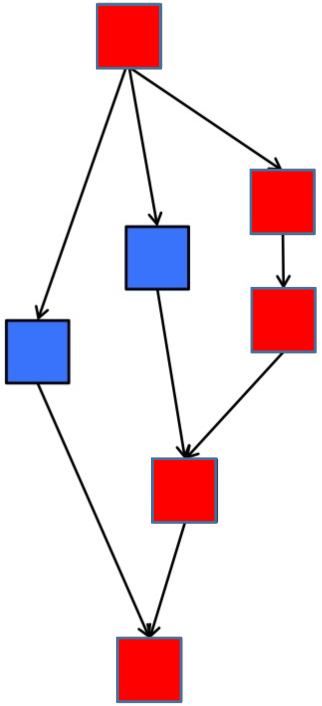
Algorithm	Work	Span	Parallelism
Quick sort	$\Theta(n \lg n)$	$\Theta(n)$	$\Theta(\lg n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(\lg^3 n)$	$\Theta(n/\lg^2 n)$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\lg n)$	$\Theta(n^3/\lg n)$
Strassen	$\Theta(n^{\lg 7})$	$\Theta(\lg^2 n)$	$\Theta(n^{\lg 7}/\lg^2 n)$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \lg n)$	$\Theta(n^2/\lg n)$
Tableau construction	$\Theta(n^2)$	$\Theta(n^{\lg 3})$	$\Theta(n^{2-\lg 3})$
FFT	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$	$\Theta(n/\lg n)$

DAG Model of Computation

- Think of a program as a directed acyclic graph (DAG) of tasks
  - A task can not execute until all the inputs to the tasks are available
  - These come from outputs of earlier executing tasks
  - DAG shows explicitly the task dependencies
- Think of the hardware as consisting of workers (processors)
- Consider a *greedy* scheduler of the DAG tasks to workers
  - No worker is idle while there are tasks still to execute



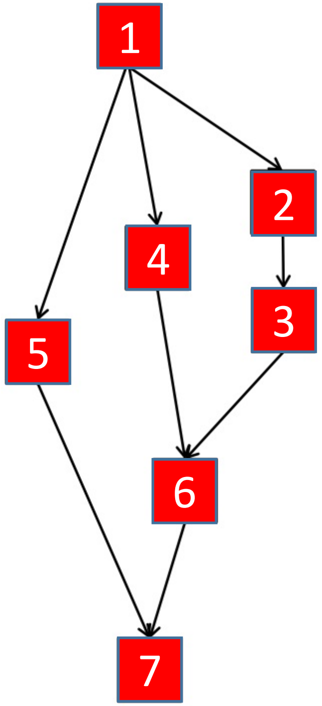
Work-Span Model



- T_P = time to run with *P* workers
- T_1 = *work*
  - Time for serial execution
    - execution of all tasks by 1 worker
  - Sum of all work
- T_∞ = *span*
  - Time along the *critical path*
- Critical path
  - Sequence of task execution (path) through DAG that takes the longest time to execute
  - Assumes an infinite # workers available

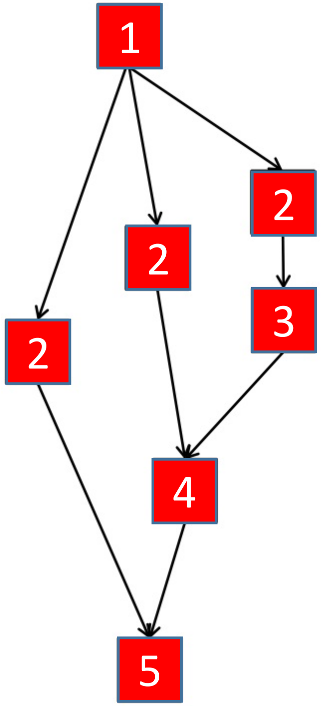
Work-Span Example

- DAG at the right has 7 tasks
- Let each task take 1 unit of time
- $T_1 = 7$
  - All tasks have to be executed
  - Tasks are executed in a serial order
  - Can them execute in any order?

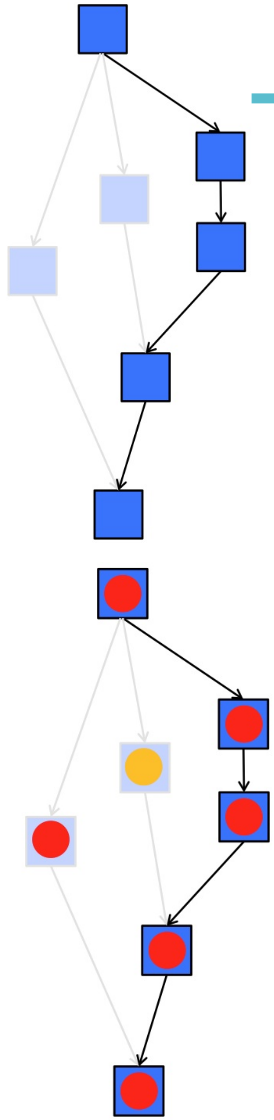


Work-Span Example

- DAG at the right has 7 tasks
- Let each task take 1 unit of time
- $T_1 = 7$
  - All tasks have to be executed
  - Tasks are executed in a serial order
  - Can them execute in any order?
- $T_\infty = 5$
  - Time along the *critical path*
  - In this case, it is the longest pathlength of any task order that maintains necessary dependencies



Lower/Upper Bound on Greedy Scheduling



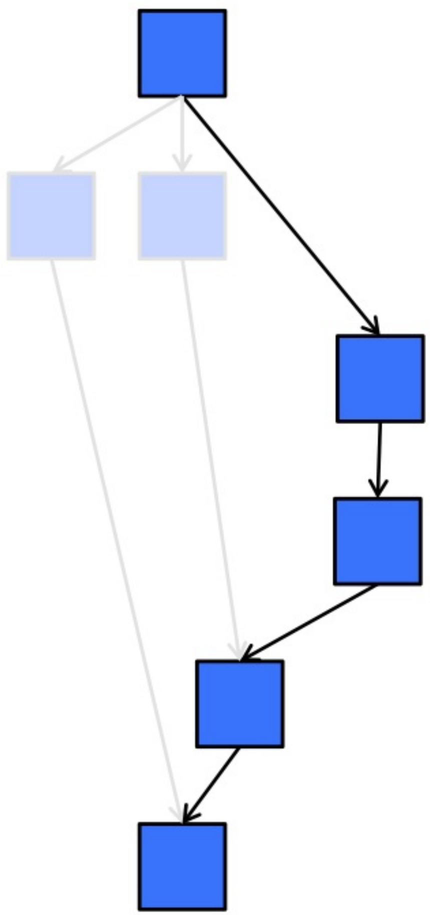
- Suppose we only have P workers
- We can write a work-span formula to derive a lower bound on T_P
  - $Max(T_1 / P , T_\infty) \leq T_P$
- T_∞ is the best possible execution time
- Brent's Lemma derives an upper bound
  - Capture the additional cost executing the other tasks not on the critical path
  - Assume can do so without overhead
  - $T_P \leq (T_1 - T_\infty) / P + T_\infty$

Consider Brent's Lemma for 2 Processors

- $T_1 = 7$
- $T_\infty = 5$
- $T_2 \leq (T_1 - T_\infty) / P + T_\infty$

  $\leq (7 - 5) / 2 + 5$

  ≤ 6

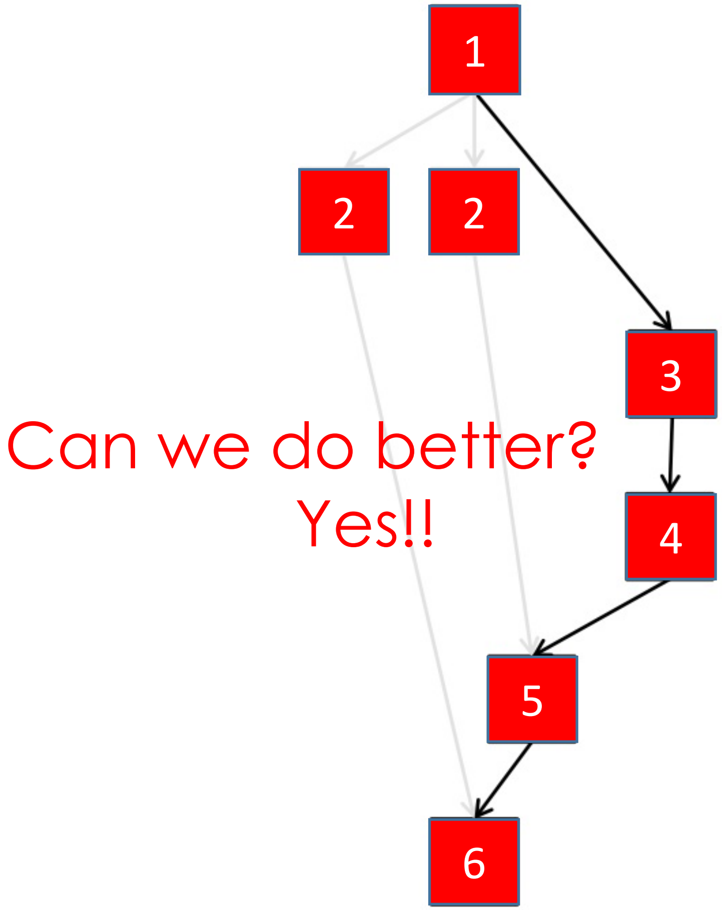


Consider Brent's Lemma for 2 Processors

- $T_1 = 7$
- $T_\infty = 5$
- $T_2 \le (T_1 - T_\infty) / P + T_\infty$

  $\le (7 - 5) / 2 + 5$

  ≤ 6

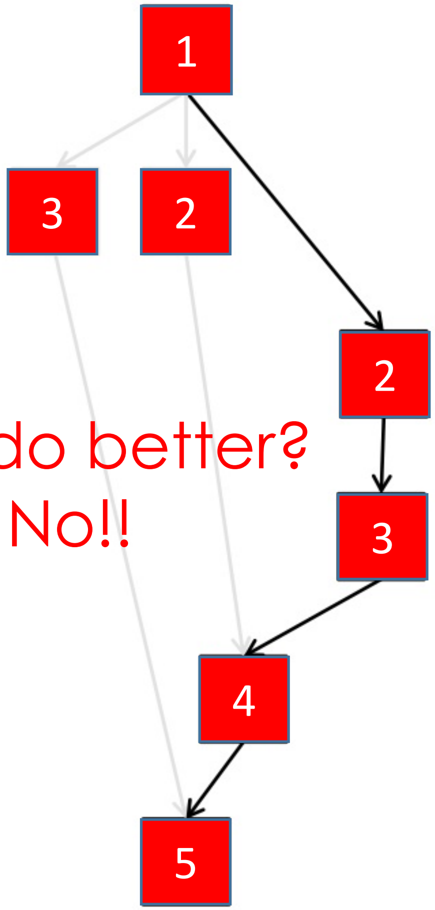


Consider Brent's Lemma for 2 Processors

- $T_1 = 7$
- $T_\infty = 5$
- $T_2 \le (T_1 - T_\infty) / P + T_\infty$

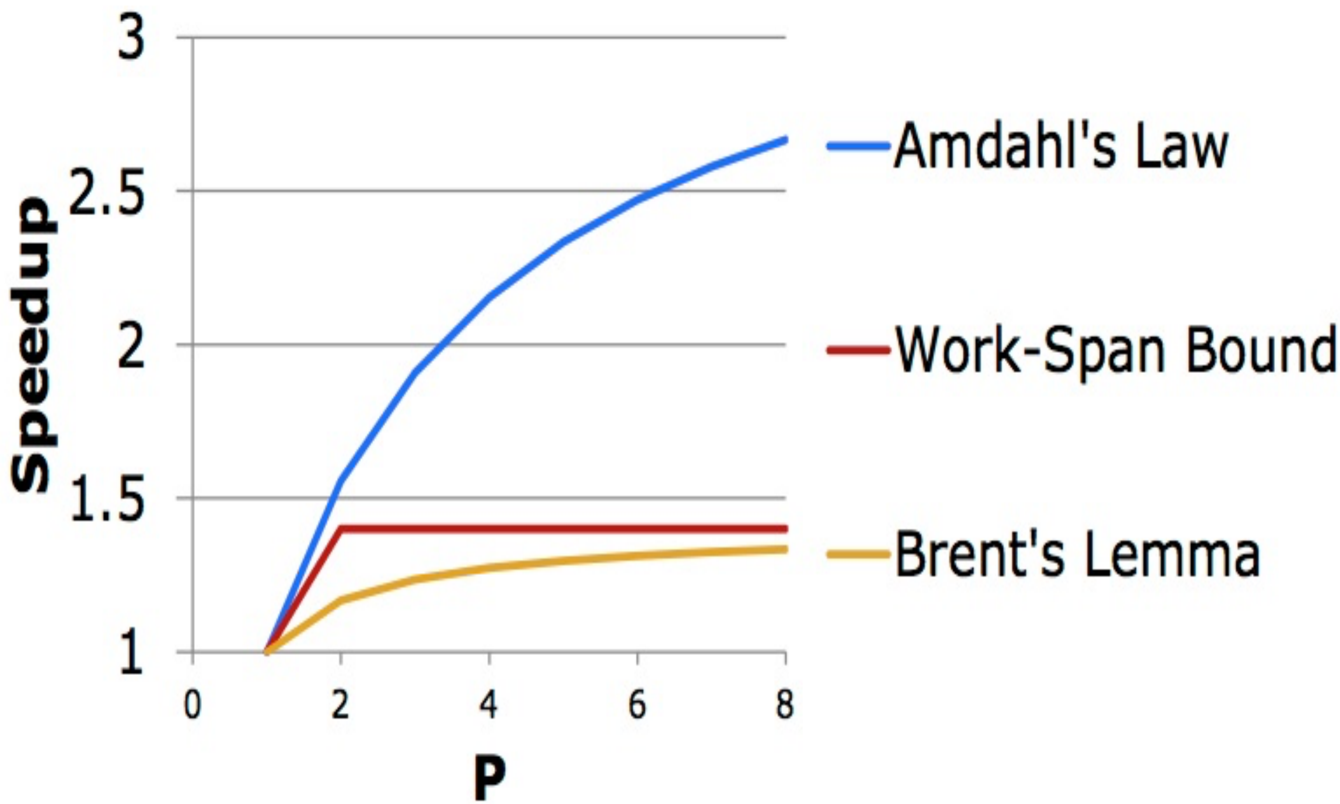
  $\le (7 - 5) / 2 + 5$

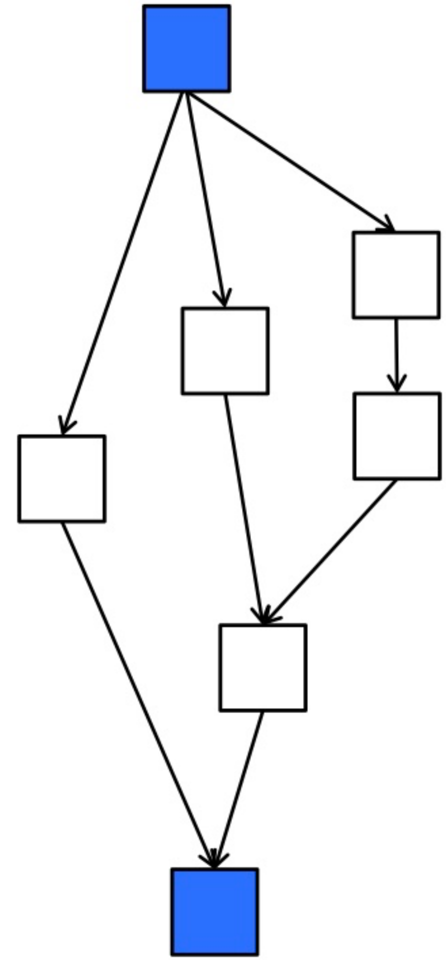
  ≤ 6



Can we do better?
No!!

Amdahl was an optimist!





Estimating Running Time

- Scalability requires that T_∞ be dominated by T_1

$$T_P \le (T_1 - T_\infty)/P + T_\infty$$

$$T_P \approx T_1/P + T_\infty \quad if \quad T_\infty << T_1$$

- Increasing work hurts parallel execution proportionately
- The span impacts scalability, even for finite P

Parallel Slack

- Sufficient parallelism implies linear speedup

$$\underbrace{T_P \approx T_1/P}_{\text{Linear speedup}} \quad if \quad \underbrace{T_1/T_\infty >> P}_{\text{Parallel stack}}$$

The END

- Sources:
  - Parallel Computing, CIS 410/510, Department of Computer and Information Science
  - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2010/video-lectures/lecture-13-parallelism-and-performance/MIT6_172F10_lec13.pdf