Algorithms and Satisfiability

Lecture 5: Parallel Algorithms

DAT6 spring 2023

Simonas Šaltenis

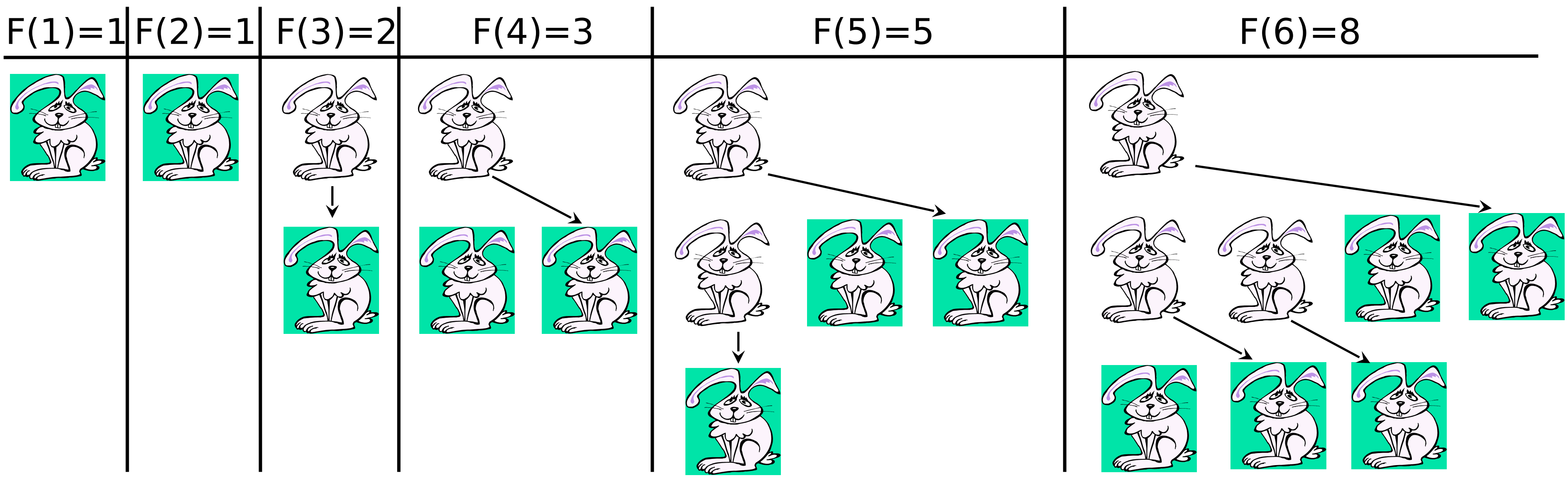
AALBORG UNIVERSITY
DENMARK

Parallel algorithms

- Goals of the lecture:
  - *to understand the model of* ***dynamic multithreading*** *(aka* ***fork-join parallelism****);*
  - *to understand* ***work****,* ***span****, and* ***parallelism*** *— the concepts necessary for the analysis of parallel algorithms;*
  - *to understand and be able to analyze the* ***parallel merge sort*** *algorithm.*

Fibonacci Numbers

- *Leonardo Fibonacci (1202)*:
  - A rabbit starts producing offspring on the second generation after its birth and produces one child each generation
  - How many rabbits will there be after *n* generations?



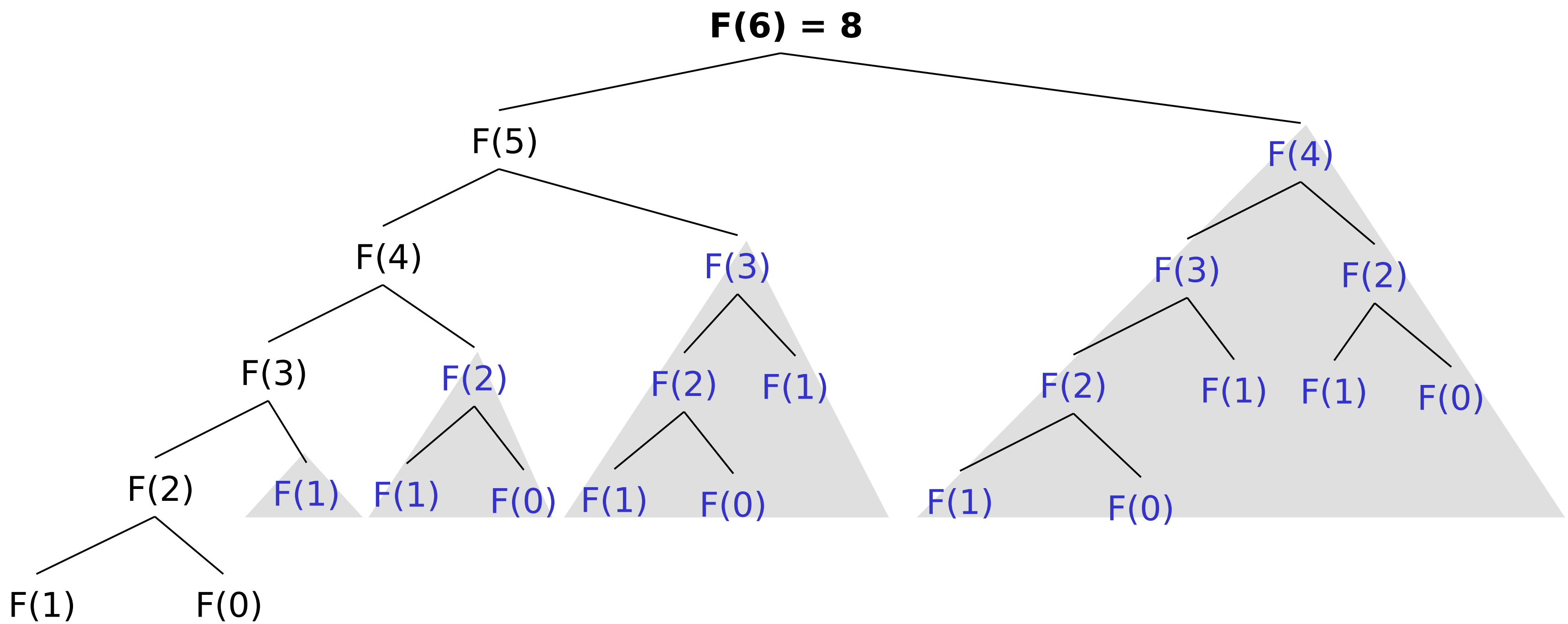
Fibonacci Numbers (2)

- *F(n)= F(n-1)+ F(n-2)*
- *F(0) =0, F(1) =1*
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 …

```
Fibonacci(n)
if n ≤ 1 then return n
else
   x = Fibonacci(n-1)
   y = Fibonacci(n-2)
return x + y
```

- Straightforward recursive procedure is slow!
- Why? How slow?
- Let’s draw the recursion tree

Fibonacci Numbers (3)



Fibonacci Numbers (4)

- How many summations are there $W(n)$?
  - $W(n) = W(n-1) + W(n-2) + 1$
  - $W(n) \geq \; 2W(n-2) + 1$ and $W(1) = W(0) = 0$
  - Solving the recurrence we get

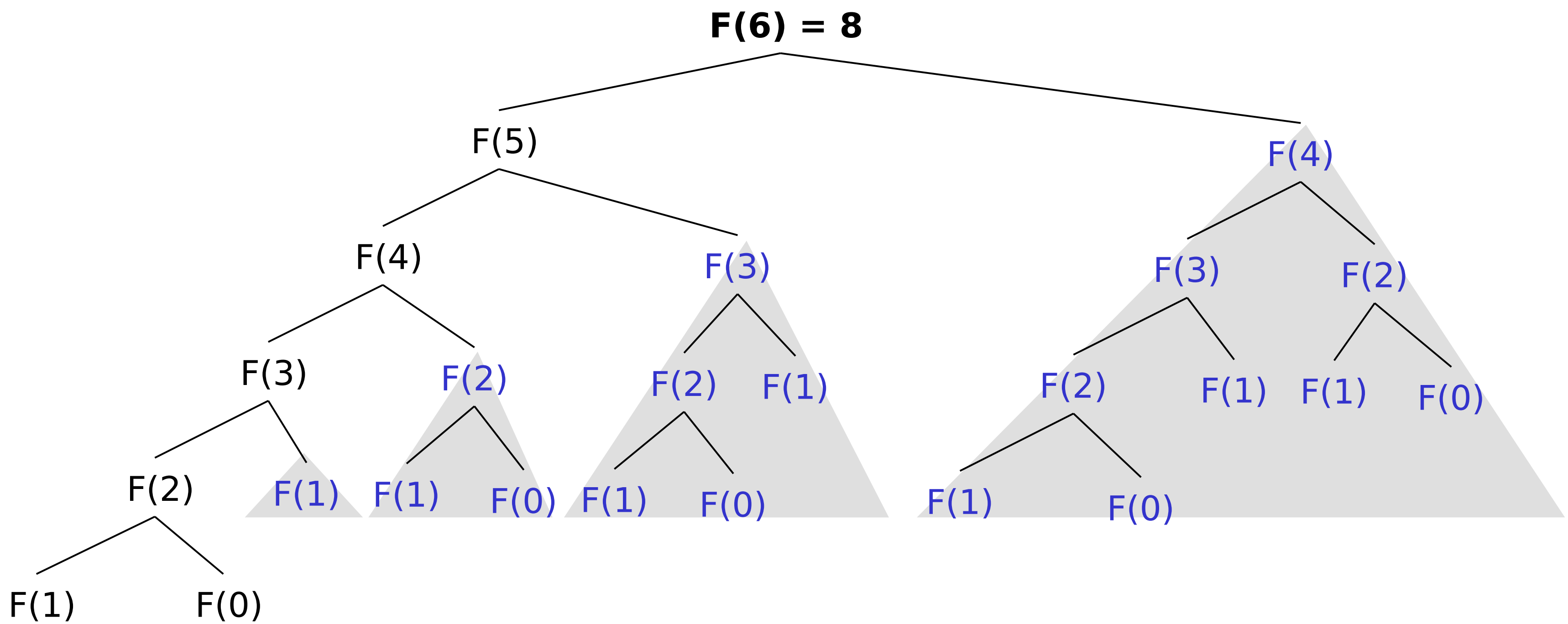
    $W(n) \geq \; 2^{(n-1)/2} - 1 \approx 1.4^{n-1}$
  - Precisely $W(n) = \Theta(\varphi^n)$, were φ is the *golden ratio* $(1+\sqrt{5})/2 \approx 1.618$
- Running time is exponential.

Multithreaded version

- What if we can do the two recursive calls in parallel
  - Using the so-called *nested parallelism*

```
FibonacciP(n)
if n ≤ 1 then return n
else
   x = spawn FibonacciP(n-1)
   y = FibonacciP(n-2)
   sync
return x + y
```

FibonacciP analysis



- "Running time":
  - $S(n) = \max(S(n-1), S(n-2)) + 1 = S(n-1) + 1$
  - Thus $S(n) = \Theta(n)$

Work, Span, Parallelism

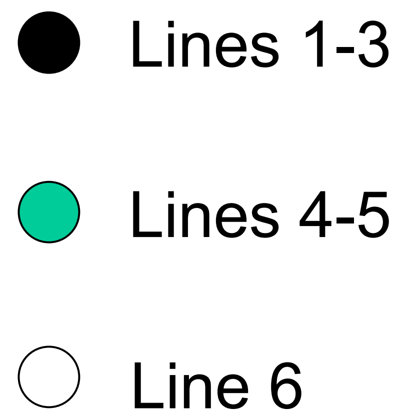
- Three main concepts (informally):
  - *Work*: the running time on a machine with one-processor (T_1).
    - Fibonacci: $\Theta(\varphi^n)$
  - *Span*: the running time on a machine with infinite processors (T_∞).
    - Fibonacci: $\Theta(n)$
  - *Parallelism* = *Work/Span* – how many processors on average are used by the algorithm.
    - Fibonacci: $\Theta(\varphi^n / n)$

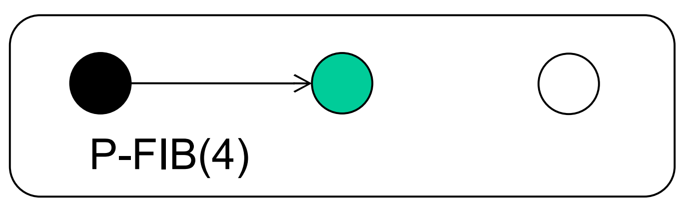
- More formally:
  - *Computation log* – a DAG of serial strands of instructions (vertices) and dependencies (edges) between them.
  - Work = the number of vertices in the computation log.
  - Span = the length of the *longest path (critical path)* in the computation log.

Computation DAG

- Using an example of computing Fibonacci number of 4.

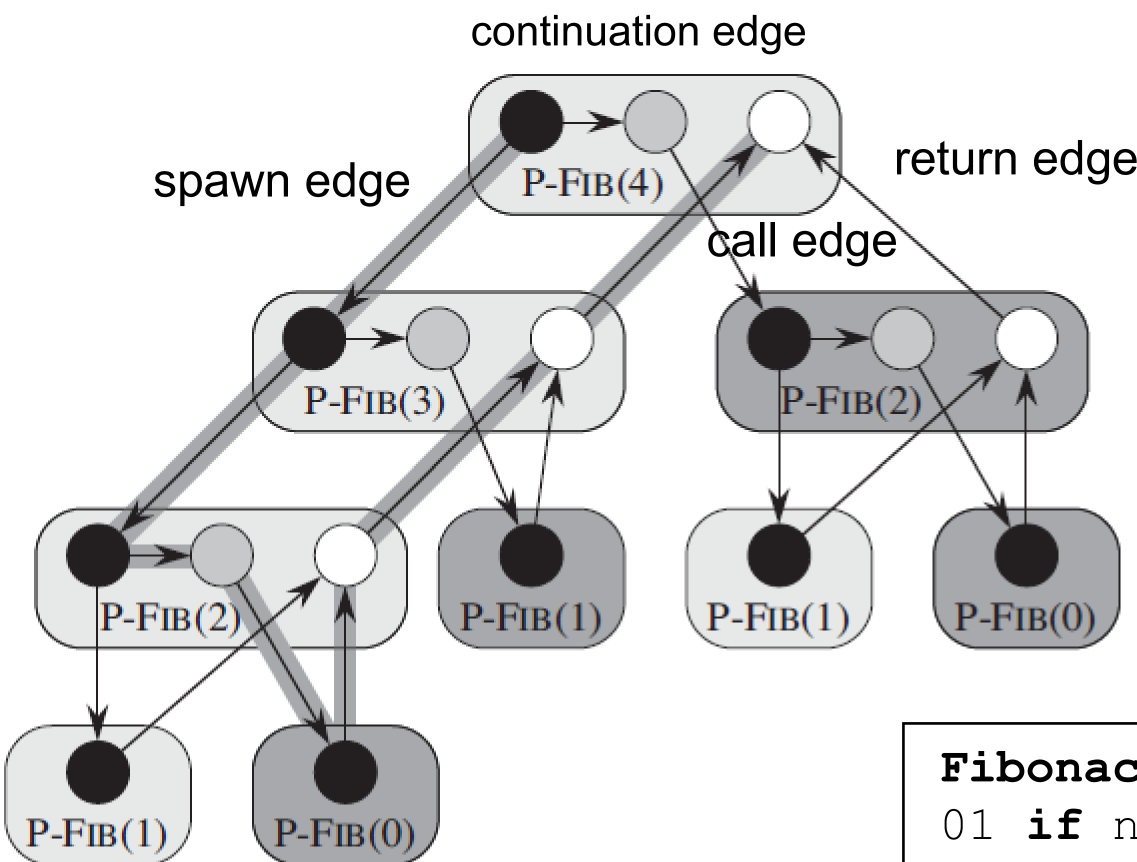
```
FibonacciP(n)
01 if n ≤ 1 then return n
02 else
03    x = spawn FibonacciP(n-1)
04    y = FibonacciP(n-2)
05    sync
06 return x + y
```

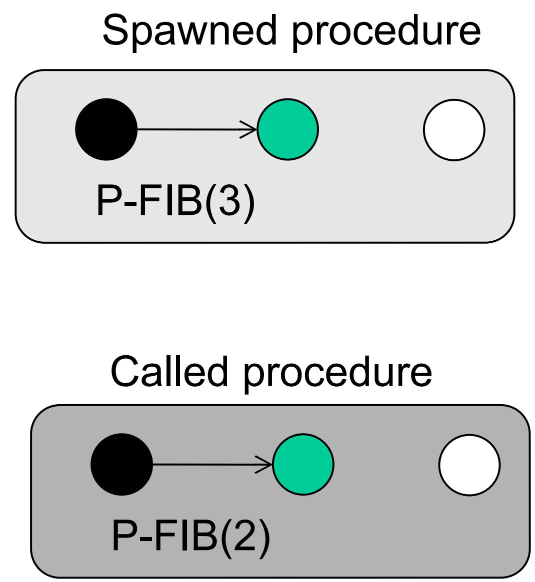




Computation DAG

Edge(u, v) means that
u must execute before v.





Work: number of vertices, 17
Span: the length of the longest path (critical path), 8

```
FibonacciP(n)
if n ≤ 1 then return n
else
   x = spawn FibonacciP(n-1)
   y = FibonacciP(n-2)
   sync
return x + y
```

Work law and span law

- Notation
  - Work T_1
  - Span T_∞
  - Multithreaded computation on P processors: T_P

- Work law: $T_P \geq T_1 / P$
  - An ideal parallel computer with P processors can do at most P units of work.
- Span law: $T_P \geq T_\infty$
  - An ideal parallel computer with P processors cannot run any faster than a machine with unlimited number of processors.

Assumptions

- The *fork-join parallelism (dynamic multithreading)* environment:
  - Shared-memory multi-core system
  - *Concurrency platform* – *task-prallel programming* :
    - Takes care of allocating work to physical threads (in other words: scheduling logical threads on physical threads)
    - Takes care of synchronization, consistent access to memory
  - Pseudocode keywords: *spawn, sync* and *parallel*
    - Indicates *potential* (or *logical*) parallelism: what *may* run in parallel.
  - We do not consider locking, race conditions, etc:
    - Parallel threads are *independent* – they work on separated items of data.
  - We abstract from actual physical scheduling:
    - It can be shown that simple greedy scheduling works well enough.

Speedup, Slackness

- When running on an actual system with *P* physical threads:
  - *Slackness* of a computation: *Parallelism / P.*
  - *What does it mean when slackness < 1? Slackness > 1?*

  - *Speedup* = T_1 / T_P.
  - *Perfect linear speedup*, when speedup = *P.*

Mini quiz

- Considering the case for computing P-Fib(4).
- We already know that the work T_1 =17 and the span T_∞ = 8
- Consider the following setups, each setup corresponds to a machine with P processing units. Which one is the most likely setup to achieve the *perfect linear speedup*?

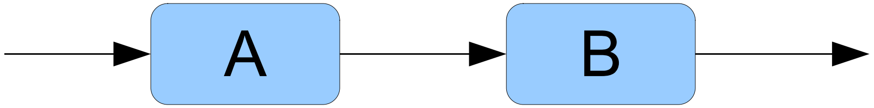
- A: P=2, B: P=3, C: P=4?
- *Go to Socrative and vote.*

To Summarize

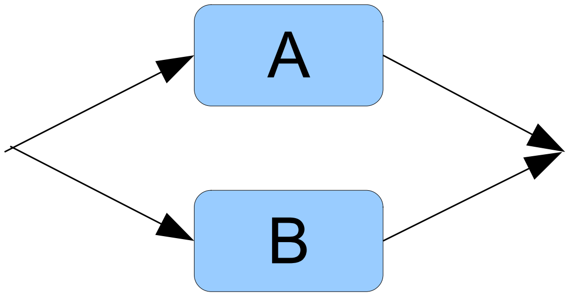
Notation	Meaning
T_1	Work, the running time on a machine with one processor.
T_∞	Span, the running time on a machine with infinite processors.
T_P	The running time on a machine with P processors.
$T_P \geq T_1 / P$	Work law
$T_P \geq T_\infty$	Span law
T_1 / T_P	Speedup. Speedup must be ≤ P according to the work law. When speedup is equal to P, it achieves ***perfect speed up***.
T_1 / T_∞	Parallelism. The maximum possible speedup that can be achieved on any number of processors
T_1 / PT_∞	Slackness = Parallelism/P. The larger the slackness, the more likely to achieve perfect speed up. When slackness is less than 1, it is impossible to achieve perfect speed up.

Computing span

- Sequential execution:
  - Work and span: $T(A\ followed\_by\ B) = T(A) + T(B)$



- Parallel execution:
  - Work: $T_1(A\ in\_parallel\_with\ B) = T_1(A) + T_1(B)$
  - Span: $T_\infty(A\ in\_parallel\_with\ B) = max(T_\infty(A), T_\infty(B))$



Goal of algorithm design

- Goal of the parallel algorithm design – *increase parallelism.*
  - Usually achieved by decreasing span (remember, parallelism = W/S)
  - It may pay off to slightly increase work, if span can be decreased significantly (in practice, relevant for highly parallel systems, such as supercomputers, but also GPUs)

- Intra-operation parallelism vs. inter-operation parallelism.

Side Note: Efficient Fibonacci

- The efficient *O*(n) serial algorithm:
  - Simple application of “*dynamic programming*” (or *memoized* evaluation of the recursive version)

```
FibonacciImproved(n)
if n ≤ 1 then return n
Fim2 ←0
Fim1 ←1
for i ← 2 to n do
  Fi ← Fim1 + Fim2
  Fim2 ← Fim1
  Fim1 ← Fi
return Fi
```

  - Can be actually done in *O*(lg *n*) additions and multiplications.

Parallel loops

- Denoted by the **parallel** keyword

```
ArrayCopy(A, B)
parallel for i = 1 to sizeof(A) do
    B[i] = A[i]
```

- Analysis of span:
  - $S(\text{n}) = O(\lg n) + \max_i \text{S\_iteration}(i)$
  - *Why*?
  - Parallel loop is implemented by divide-and-conquer

```
ArrayCopyRecursive(A, B, l, r)
if l > r then return
if l = r then B[l] = A[r]
else
   q = ⌈(l+r)/2⌉
   spawn ArrayCopyRecursive(A, B, l, q-1)
         ArrayCopyRecursive(A, B, q, r)
```

Examples

- Exchanging neighboring elements:

```
ArrayExchange(A)
parallel for i = 1 to ⌊sizeof(A)/2⌋ do
   tmp       = A[i*2]
   A[i*2]    = A[i*2-1]
   A[i*2-1] = tmp
```

- Compute the largest stock price difference :

```
LargestSpike(A)
lspike = 0
parallel for i = 1 to sizeof(A)-1 do
   if |A[i+1] - A[i]| > lspike then
      lspike = |A[i+1] - A[i]|
return lspike
```

Merge Sort

```
Merge-Sort(A, p, r)
if p < r then
    q = ⌊(p+r)/2⌋
    Merge-Sort(A, p, q)
    Merge-Sort(A, q+1, r)
    Merge(A, p, q, r)
```

Divide

Conquer

Combine

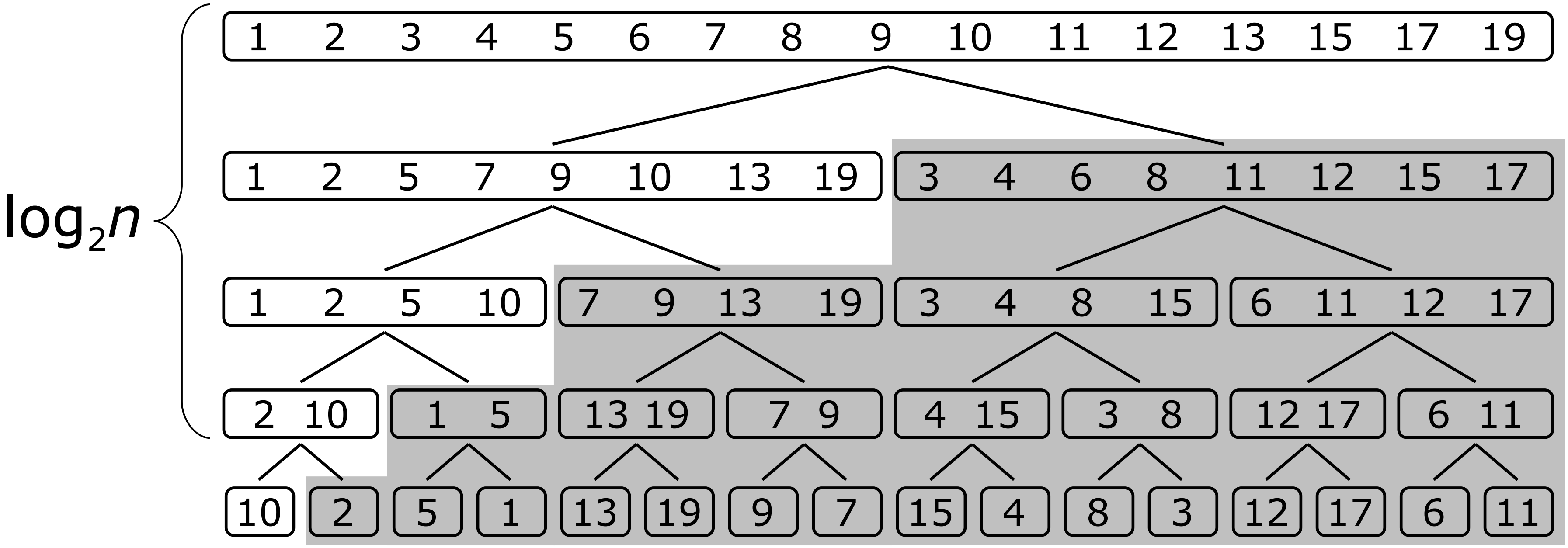
- Running time?

Parallelising Merge Sort

```
Merge-Sort'(A, p, r)
if p < r then
    q = ⌊(p+r)/2⌋
    spawn Merge-Sort'(A, p, q)
    Merge-Sort'(A, q+1, r)
    sync
    Merge(A, p, q, r)
```

- Work
  - $W(n) = 2W(n/2) + \Theta(n)$
  - $W(\mathrm{n}) = \Theta(n \lg n)$
- Span:
  - $S(n) = S(n/2) + \Theta(n)$
  - $S(\mathrm{n}) = \Theta(n)$
- Parallelism: $W(n)/S(n) = \Theta(\lg n)$. Rather low...

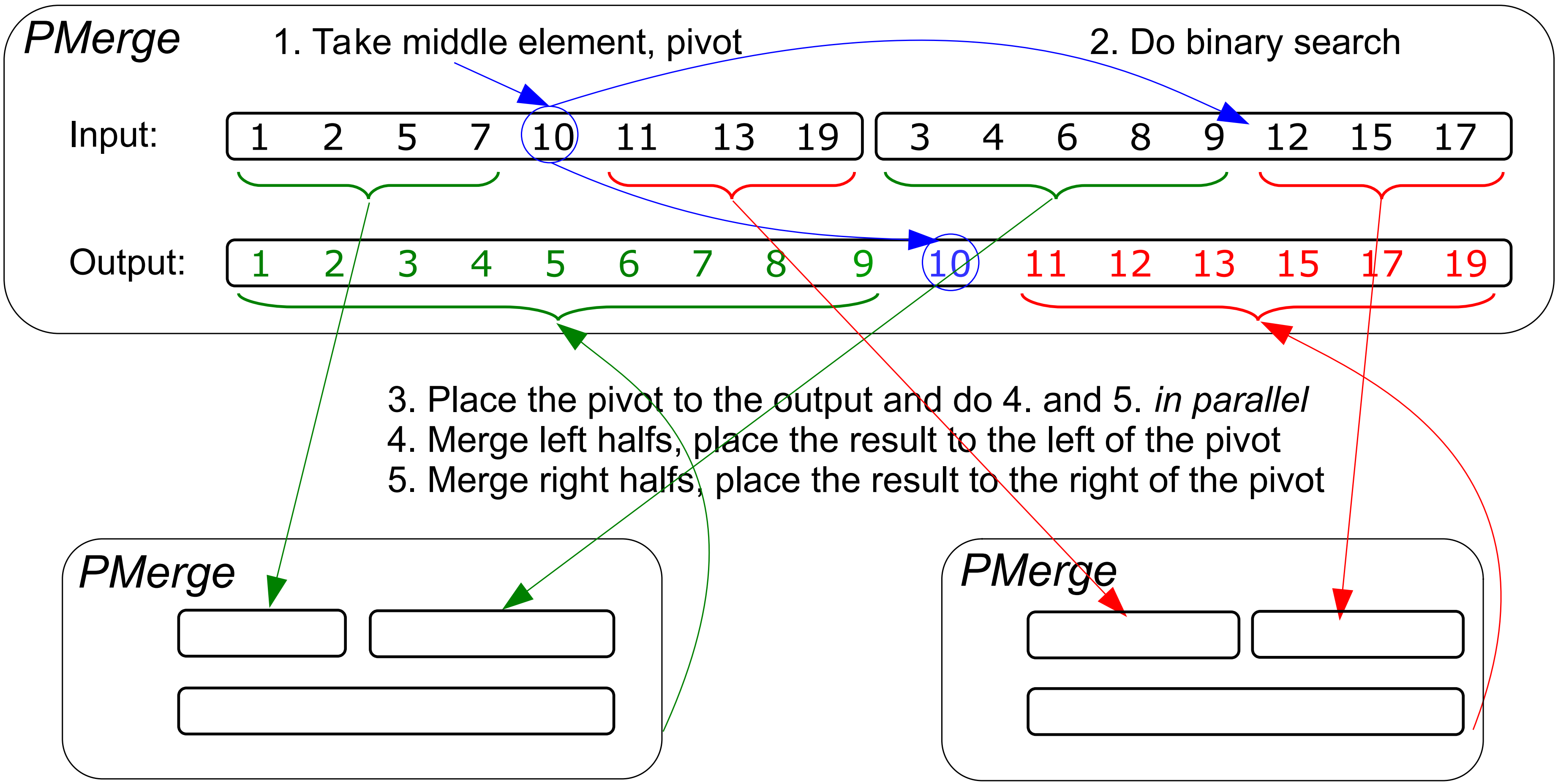
Merge-Sort Recursion Tree



- Problem – merging is very serial:
  - At the top level, only one processor does $\Theta(n)$ work in serial!
  - At the second level, only two processors do $\Theta(n)$ work.
  - …

Multithreaded merging

- Main idea – make the algorithm divide-and-conquer and use nested parallelism.



Multithreaded merging analysis

- One key idea: do binary search in the smaller of the two arrays!
  - This ensures that the *largest* of the two recursive calls works with *at most* $3n/4$ elements, where n = the sum of sizes of the two arrays.
  - *Why?*
- Span:
  - $S(n) = S(3n/4) + \Theta(\lg n)$
  - *What is the solution*?
  - $S(n) = \Theta(\lg^2 n)$
- Work:
  - Can be shown to be $\Theta(n)$.

Multithreaded merge sort

```
PMerge-Sort(A, p, r)
if p < r then
    q = ⌊(p+r)/2⌋
    spawn PMerge-Sort(A, p, q)
    PMerge-Sort(A, q+1, r)
    sync
    PMerge(A, p, q, r)
```

- Work:
  - The same recurrence and solution: $W(\mathrm{n}) = \Theta(n \lg n)$
- Span:
  - $S(n) = S(n/2) + \Theta(\lg^2 n)$
  - $S(n) = \Theta(\lg^3 n)$
- Parallelism:
  - $W(n) / S(n) = \Theta(n \lg n) / \Theta(\lg^3 n) = \Theta(n / \lg^2 n)$

To summarize

- Work: $\Theta(n \lg n)$

	Merge Procedure	Span	Parallelism
Naïve merge	$\Theta(n)$	$\Theta(n)$	$\Theta(\lg n)$
P-Merge	$\Theta(\lg^2 n)$	$\Theta(\lg^3 n)$	$\Theta(n / \lg^2 n)$

Goal of the multi-threaded algorithm design

- Goal of the multi-threaded algorithm design – *increase parallelism.*
  - *Parallelism = work / span*
  - Usually achieved by decreasing span
    - MergeSort without P-Merge and with P-Merge
    - $\Theta(n)$ vs. $\Theta(lg^3 n)$
  - It may pay off to slightly increase work, if span can be decreased significantly (in practice, relevant for highly parallel systems, such as supercomputers, GPUs)