Advanced Algorithms

Lecture 10
Multithreaded Algorithms

Tung Kieu
tungkvt@cs.aau.dk

ILO of Lecture 10

- Multithreaded algorithms
  - to understand the model of dynamic multithreading, including **nested parallelism** and **parallel loops**;
  - to understand **work, span,** and **parallelism** — the concepts necessary for the analysis of multithreaded algorithms;
  - to understand and be able to analyze the multithreaded merge sort algorithm.

Agenda

- Background
- Nested parallelism
- Work, span, and parallelism
- Parallel loops
- Multithreaded merge sort

Parallel computers

- Computers with multiple processing units
  - Chip multiprocessors / Low price
    - A single multi-core chip has multiple
    - My laptop: Intel® Core™ i7-7700U
  - Clusters / Intermediate price.
    - Individual computers that are conne
  - Supercomputers / High price.
    - Combination of custom architectures
      the highest performance.

- Memory models
  - Shared memory: all processors can
    memory.
  - Distributed memory: where each processor has a private memory.
  - No agreement on a single architectural model for parallel computers so far.

Static vs. dynamic threading

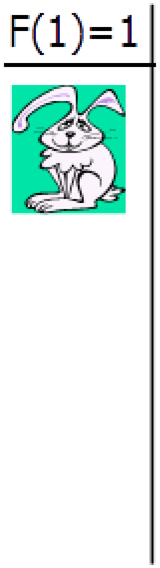
- Static threading
  - Each thread maintains an associated program counter and executes code independently of the other threads.
  - Threads persist for the duration of a computation.
  - Directly using static threading is difficult and error-prone.
- Dynamic threading
  - Specify parallelism in applications without worrying implementation details.
  - A ***concurrency platform's scheduler*** does communication, load balancing, etc.
  - ***Nested parallelism and parallel loops.***

Agenda

- Background
- Nested parallelism
- Work, span, and parallelism
- Parallel loops
- Multithreaded merge sort

Fibonacci Numbers

- *Leonardo Fibonacci (1202)*:
  - We have a rabbit in the beginning.
  - 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?



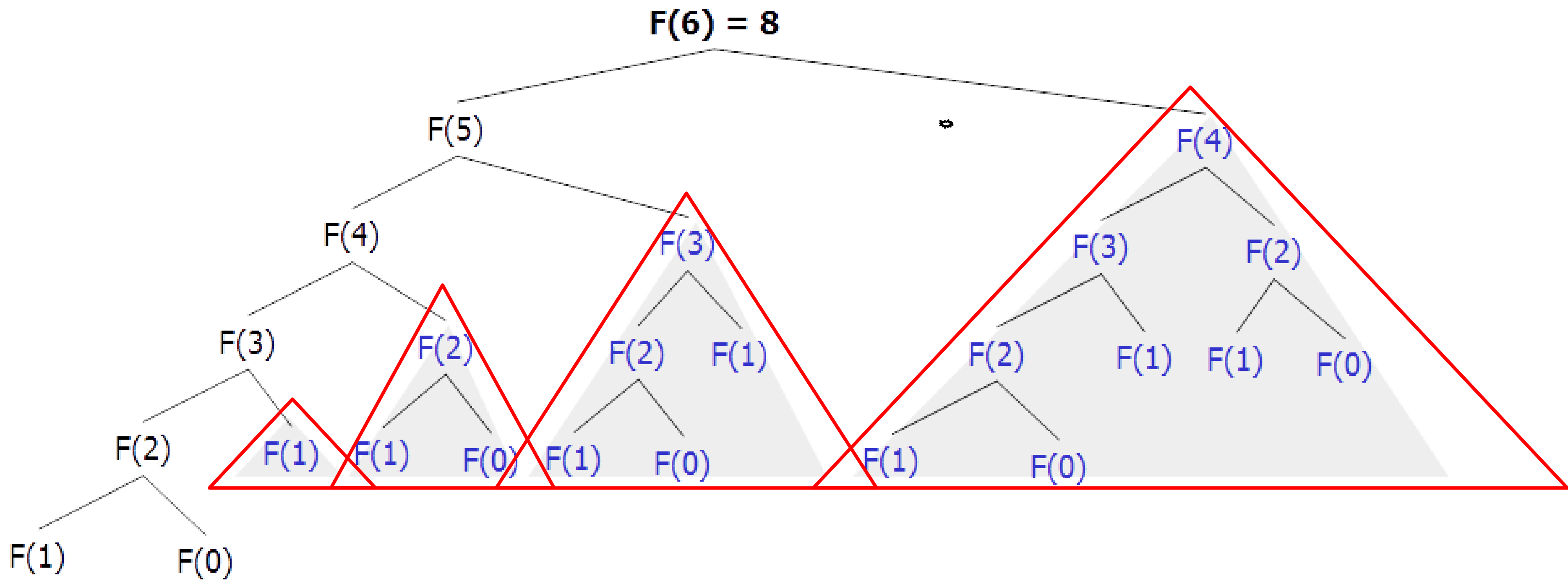
Fibonacci Numbers (2)

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

```
FIB(n)
if n ≤ 1
        return n
else x = FIB(n − 1)
        y = FIB(n − 2)
        return x + y
```

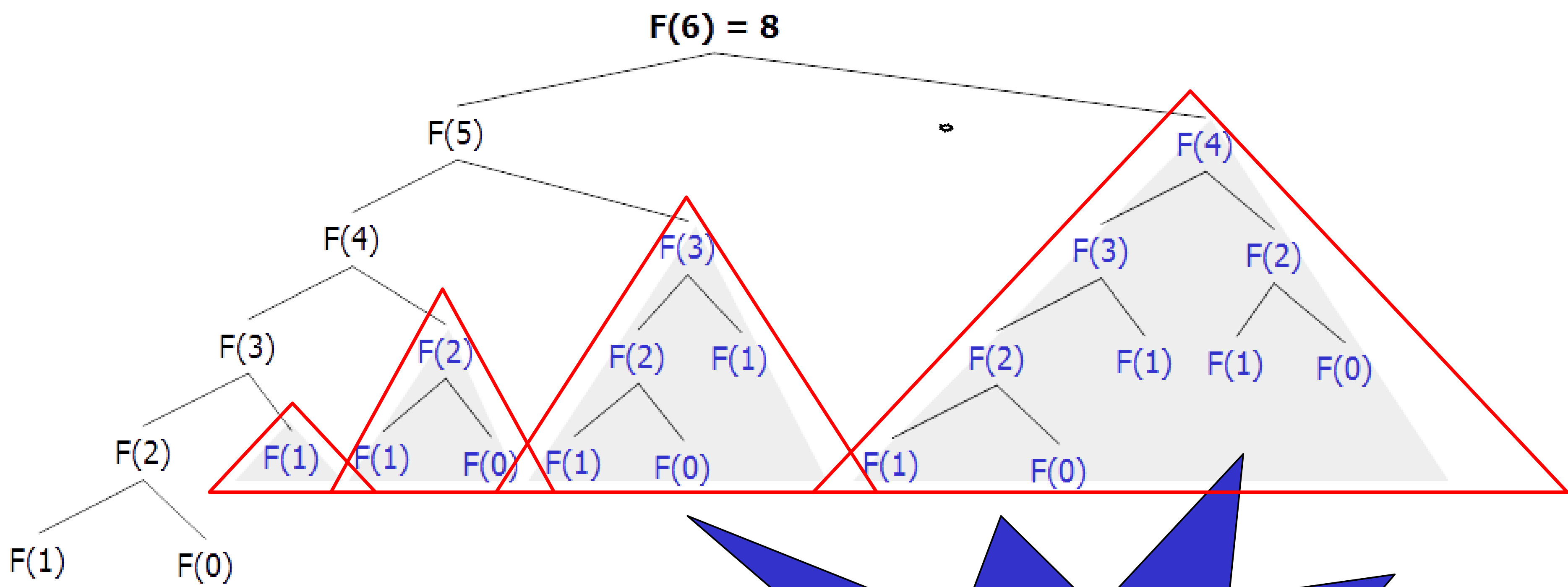
- Straightforward recursive procedure is very very slow!
- Why? How slow? $O(2^n)$
- Let’s draw the recursion tree.

Fibonacci Numbers (3)



- We keep calculating the same value over and over!
  - Sub-problems are overlapping – they share sub-sub-problems.
- Do you still remember how we can avoid calculating the same sub-problems?

Fibonacci Numbers (3)



- We keep calculating the s
  - Sub-problems are overlapping –
- Do you still remember how we ca the same sub-problems?

Dynamic Programming
Memoization (Top Down)
Tabulation (Bottom Up)

Fibonacci Numbers (4)

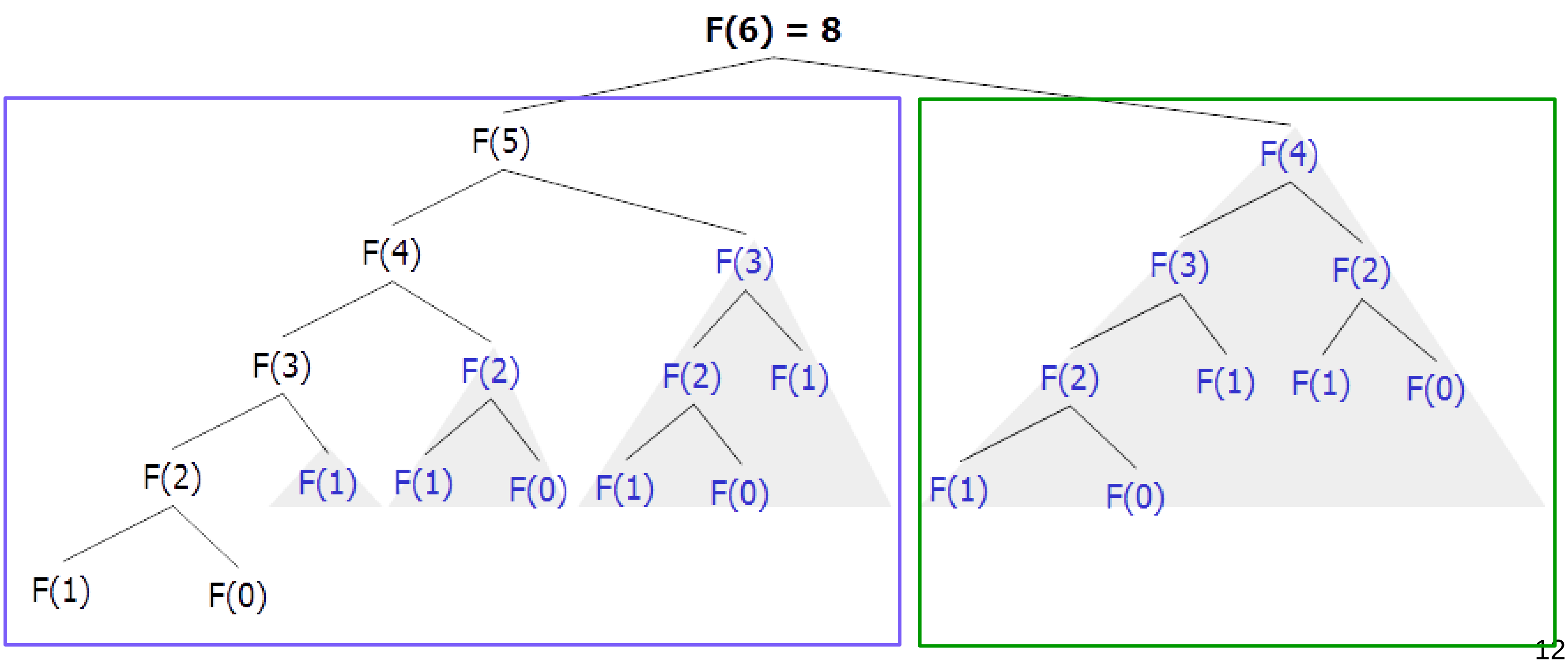
- Recurrence
  - $\boxed{T(n) = T(n-1) + T(n-2) + \Theta(1)}$ Master theorem
- Since $T(n-1) \geq T(n-2)$, we have $T(n) \geq 2T(n-2) + a$
- Solving the recurrence using the repeated substitution method.
  - $T(n)=2\boxed{T(n-2)}+a$ $\quad T(n-2)=2T(n-4)+a$
  - $T(n)=2^2\boxed{T(n-4)}+\cancel{(2+1)a}$ $\quad T(n-4)=2T(n-6)+a$
  - $T(n)=2^3\boxed{T(n-6)}+(2^2+2+1)a$
  - $T(n)= 2^iT(n-2*i)+(2^{i-1}+\ldots+2+1)a = 2^iT(n-2*i)+a\sum_{k=0}^{i-1} 2^k$
  - When $i=n/2$, $T(n)=2^{n/2}\,T(0) + a*\,2^{n/2}= (a+1)\,2^{n/2}$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

- We get
  - $T(n) \geq (a+1)\,2^{n/2} \approx (a+1)\,1.4^n$
- Running time is ***exponential!***

Multithreaded version

- What if we can do the two recursive calls in parallel?
- Using the so-called *nested parallelism*
  - Call a procedure. Don’t wait for it to return and go to next step.
  - Call Fib(5), we do not need to wait for Fib(5) returns results and we can call Fib(4).



Concurrency keywords

- Spawn
  - If the keyword **spawn** precedes a procedure call, it indicates a ***nested parallelism***.
  - No need to wait for the procedure with keyword **spawn**.
    - We do not wait P-FIB(n-1) to complete and we can execute P-FIB(n-2) on another processing unit.
- Sync
  - Must wait for all spawned procedures to complete before going to the statement after **sync**.
    - Before “return x+y”, we must wait P-FIB(n-1) to complete.

```
P-FIB(n)
if n ≤ 1
    return n
else x = spawn P-FIB(n − 1)
    y = P-FIB(n − 2)
    sync
    return x + y
```

```
FIB(n)
if n ≤ 1
    return n
else x = FIB(n − 1)
    y = FIB(n − 2)
    return x + y
```

Analysis

- Recurrence:
  - $T(n) = \max(T(n-1), T(n-2)) + c = T(n-1) + c$
  - Can you try to solve this recurrence?
- Solving the recurrence, we have $T(n) = \Theta(n)$.
- The serial version is ***exponential*** whereas the multithreaded version is only ***linear***.

$$
\begin{aligned}
&T(n) = T(n-1) + c \\
&T(n-1) = T(n-2) + c \\
&\Rightarrow T(n) = (T(n-2) + c) + c = T(n-2) + 2c \\
&T(n-2) = T(n-3) + c \\
&\Rightarrow T(n) = T(n-3) + 3c \\
&\Rightarrow \ldots \\
&\Rightarrow T(n) = T(0) + nc
\end{aligned}
$$

Agenda

- Background
- Nested parallelism
- Work, span, and parallelism
- Parallel loops
- Multithreaded merge sort

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, work and span are defined using a computation DAG (directed acyclic graph).
  - Vertices are instructions or sets of instructions.
  - Edges represent dependencies between instructions.
    - An edge (u, v) means that u must execute before v.

Computation DAG

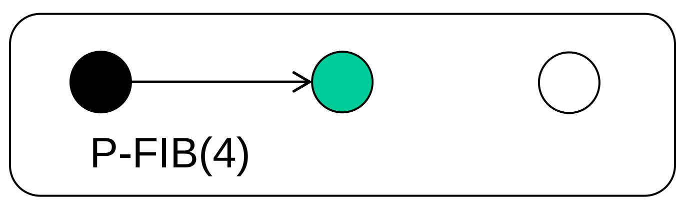
- Using an example of computing Fibonacci number of 4.

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)
4      y = P-FIB(n − 2)
5      sync
6      return x + y
```

Lines 1-3

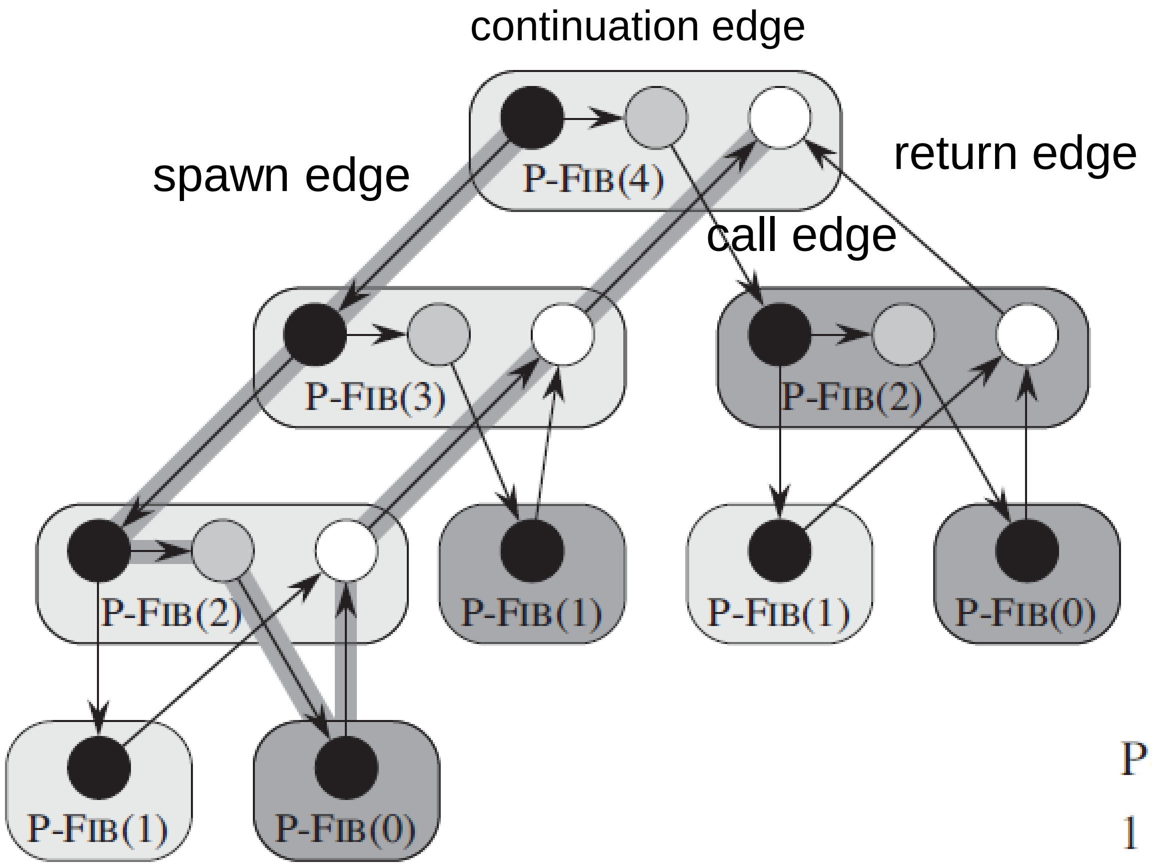
Lines 4-5

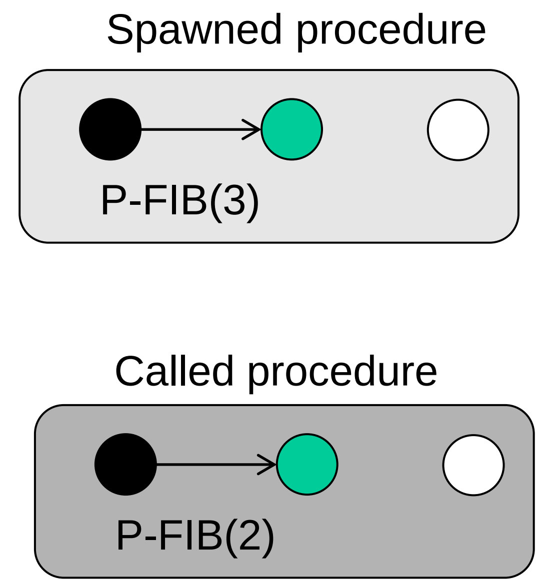
Line 6



Computation DAG

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





P-FIB(n)

```
if n ≤ 1
    return n
else x = spawn P-FIB(n − 1)
    y = P-FIB(n − 2)
    sync
    return x + y
```

Work: number of vertices, 17

Span: the length of the longest path (critical path), 8

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.

Ex: $O(n^2) \rightarrow O(n)$ if we have n processor

Speedup

- Speedup: T_1/T_P:
  - How many times faster the computation on P processors than on a single processor.
  - Work law
  - Recall the work law says that $\boxed{T_P \geq T_1 / P,}$ so that the speedup must be smaller than or equal to P.
    - The speedup on a P-processor machine can ***at most*** be P.
    - If the speedup is P, we have ***perfect linear speedup***.

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***?

- P=2, P=3, P=4, P=5, P=10

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***?
- P=2, P=3, P=4, P=5, P=10
- Span law: $T_P \geq T_\infty$
  - P=2, having perfect linear speedup indicates $T_2 = 17/2 = 8.5 \geq T_\infty$, which is possible.
  - P=3, having perfect linear speedup indicates $T_3 = 17/3 = 5.7 \leq T_\infty$, which is impossible.
  - So for P=4, P=5, and for any P if P is greater than 2.

Parallelism and Slackness

- Parallelism: T_1 / T_∞ :
  - The maximum possible speedup that can be achieved on any number of processors.
  - Once the number of processors exceeds the parallelism, the computation cannot possibly achieve the perfect linear speedup.
- Slackness: Parallelism/P = T_1 / PT_∞
  - Assume that P increases from 1 each time by 1.
  - Slackness drops from Parallelism to 1 and then from 1 to 0.
  - Slackness >=1: closer to perfect linear speedup.
  - Slackness <1: diverges further away from perfect linear speedup.
- Example: parallelism = 10.
  - P=2, slackness=5, perfect linear speedup is possible.
  - P=10, slackness=1, perfect linear speedup is possible but more difficult than when P=2
  - P=20, slackness=0.5, impossible to achieve perfect linear speedup, i.e., 20 speed up, due to the parallelism=10

Parallelism and Slackness

- Parallelism: T_1 / T_∞ :
  - The maximum possible speedup that can be achieved on any number of processors.
  - Once the number of processors exceeds the parallelism, the computation cannot possibly achieve the perfect linear speedup.
- Slackness: Parallelism/P = [illegible]
  - Assume that P increas[illegible]
  - Slackness drops from Paral[illegible]
  - Slackness >=1: closer to [illegible]
  - Slackness <1: diverges further a[illegible] linear speedup.
- Example: parallelism = 10.
  - P=2, slackness=5, perfect linear speedup is possible.
  - P=10, slackness=1, perfect linear speedup is possible but more difficult than when P=2
  - P=20, slackness=0.5, impossible to achieve perfect linear speedup, i.e., 20 speed up, due to the parallelism=10

17/8

17/P8

P=1 → 17/8

P=2 → 17/16

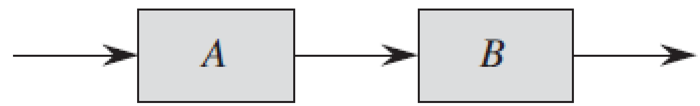
P=3 → 17/24

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.

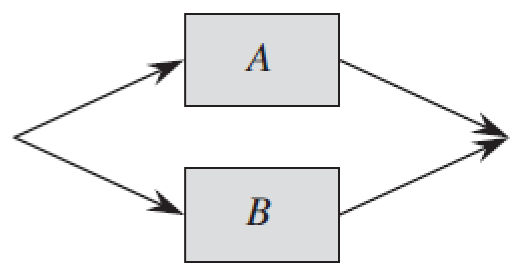
Analyzing multithreaded algorithms

- We have two sub-computations A and B.
- If they are joined in series:



$$\text{Work: } T_1(A \cup B) = T_1(A) + T_1(B)$$
$$\text{Span: } T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$$

- If they are joined in parallel:



$$\text{Work: } T_1(A \cup B) = T_1(A) + T_1(B)$$
$$\text{Span: } T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$$

Analyzing multithreaded Fibonacci

- $T_\infty(n) = \max(\boxed{T_\infty(n-1), T_\infty(n-2)}) + c$
- $T_\infty(n) = \boxed{T_\infty(n-1)} + c$

$T_\infty(n-2) \le T_\infty(n-1)$

- ...

$T_\infty(n-1) = \max(T_\infty(n-2) + T_\infty(n-3)) + c$

- $T_\infty(n) = T_\infty(0) + nc$
- $T_\infty(n) = \Theta(n)$.

- Parallelism of P-FIB(n) is $T_1(n)/ T_\infty(n) = \Theta(\varphi^n / n)$
  - As n increases, the parallelism grows.

Agenda

- Background
- Nested parallelism
- Work, span, and parallelism
- Parallel loops
- Multithreaded merge sort

Concurrency keywords: parallel

- **Parallel** for parallel loops
  - Just like a for loop but loop executions run concurrently

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

  - Implemented in a divide-and-conquer manner by using spawn.

```
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)
```

Here, we only make 1 spawn.
What if we make 2 spawns?
And what if we make n-1 spawns?

Span: $T_\infty(n) = \max(T_\infty(n/2), T_\infty(n/2)) = T_\infty(n/2) + c = \Theta(\lg n)$

Work: $T_1(n) = 2T_1(n/2) + c = \Theta(n)$

Why not spawn n-1 times?

- Creating a thread is not **free,** but takes some time.
  - Assume that it takes constant time **c**.
- Implemented in a divide-and-conquer manner by using one (or a constant number of) spawn(s).

```
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)
```

- Using one spawn
  - Span: $T_\infty(n) = \max(T_\infty(n/2), T_\infty(n/2))$ **+ c**
  - $T_\infty(n) = T_\infty(n/2)$ **+ c**
  - $\Theta(\lg n)$
  - Work: $T_1(n) = T_1(n/2) + T_1(n/2)$
  - $T_1(n) = 2T_1(n/2)$ **+ c**
  - $\Theta(n)$

Why not spawn n-1 times?

- Creating a thread is not **free,** but takes some time.
  - Assume that it takes constant time **c**.

- Using two spawns
  - Span: $T_\infty(n) = \max(T_\infty(n/3), T_\infty(n/3), T_\infty(n/3))$ **+ 2c**
  - $T_\infty(n) = T_\infty(n/3)$ **+ 2c**
  - $\Theta(\lg n)$
  - Work: $T_1(n) = 3T_1(n/3)$ **+ 2c**
  - $\Theta(n)$

Why not doing n copies in parallel?

- Creating a thread is not **free** but takes some time.
  - Assume that it takes constant time c.

- If we make **n-1** spawn threads and each thread copies one element, then we have the following:
  - Span: $T_\infty(n) = \max(T_\infty(n/\mathbf{n}), \ldots, T_\infty(n/\mathbf{n})) + (\mathbf{n-1})*\mathbf{c} =$
  - $T_\infty(n) = a + (\mathbf{n-1})*\mathbf{c}$
  - $\Theta(n)$
  - Work: $T_1(n) = nT_1(1) + (\mathbf{n-1})*\mathbf{c}$
  - $\Theta(n)$

Why not doing n copies in parallel?

- Creating a thread is not **free** but takes some time.
  - Assume that it takes constant time c.

- Assume that we have P processing units, and we make P-1 spawn threads.
  - Span: $T_P(n) = \max(T_P(n/\boldsymbol{P}), \ldots, T_P(n/\boldsymbol{P})) + (\mathbf{P-1})\mathbf{*c}$
  - $T_P(n) = T_P(n/\mathbf{P})+(\mathbf{P-1})\mathbf{*c}$
  - $\Theta(\mathbf{P}\lg_{\mathbf{P}} n)$
  - Assume that P=2, $T_P(n) = \Theta(2\lg_2 n)=\Theta(\lg n)$
  - Assume that P=3, $T_P(n) = \Theta(3\lg_3 n)=\Theta(\lg n)$
  - Assume that P=n, $T_P(n) = \Theta(n\lg_n n)=\Theta(n)$

Race conditions

- Race conditions happen when
  - Two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

RACE-EXAMPLE( )

$x = 0$
parallel for $i = 1$ **to** 2
    $x = x + 1$
print x

- How to avoid?
  - Need to make sure that the two threads are independent - no writing to memory that other threads are reading from.

Matrix-vector multiplication

A n*n matrix multiplies a n*1 vector. We get a n*1 vector.

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

i: row
j: column

$$\begin{bmatrix} a11 & a12 & \ldots & a18 \\ a21 & a22 & \ldots & a28 \\ \ldots & \ldots & \ldots & \ldots \\ a81 & a82 & \ldots & a88 \end{bmatrix} \quad \begin{bmatrix} x1 \\ x2 \\ \ldots \\ x8 \end{bmatrix}$$

MAT-VEC(A, x)

$n = A.rows$
let y be a new vector of length n
parallel for $i = 1$ **to** n
  $y_i = 0$
parallel for $i = 1$ **to** n
  for $j = 1$ **to** n
    $y_i = y_i + a_{ij} x_j$
return y

Can we make this also a parallel loop?

- The first parallel loop: same with ArrayCopy that we just saw.
- The second parallel loop: a bit complicated.
  - Still use D&C and Spawn.
  - Base case is the for loop with j.

Matrix-vector multiplication

A n*n matrix multiplies a n*1 vector. We get a n*1 vector.

O(n²)

$$\sum_{j=1}^{n} a_{ij} x_j$$

i: row
j: column

$$\begin{bmatrix} a11 & a12 & \dots & a18 \\ a21 & a22 & \dots & a28 \\ \dots & \dots & \dots & \dots \\ a81 & a82 & \dots & a88 \end{bmatrix} \quad \begin{bmatrix} x1 \\ x2 \\ \dots \\ x8 \end{bmatrix}$$

```
MAT-VEC(A, x)
n = A.rows
let y be a new vector of length n
parallel for i = 1 to n
    y_i = 0
parallel for i = 1 to n
    for j = 1 to n
        y_i = y_i + a_ij x_j
return y
```

Can we make this also a parallel loop?

- The first parallel loop: same with ArrayCopy that we just saw.
- The second parallel loop: a bit complicated.
  - Still use D&C and Spawn.
  - Base case is the for loop with j.

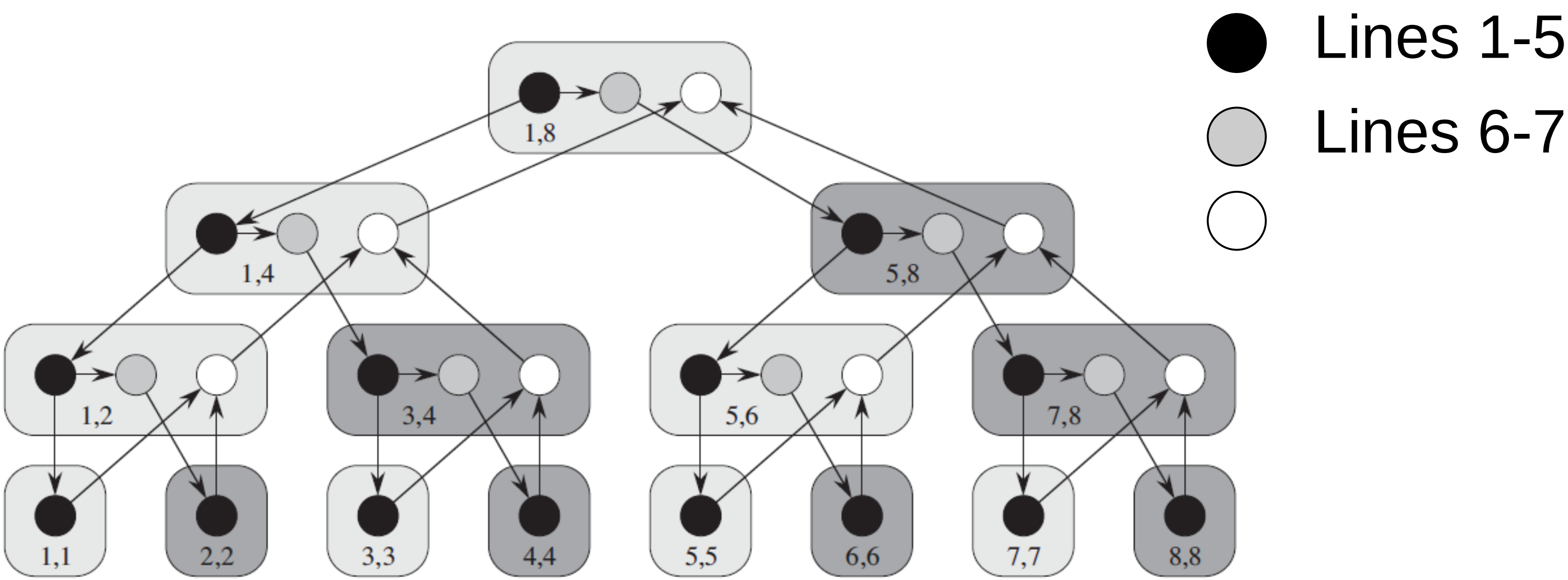
The second parallel loop

5 **parallel for** $i = 1$ **to** n
6 **for** $j = 1$ **to** n
7 $y_i = y_i + a_{ij} x_j$

MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')

1 **if** $i == i'$
2 **for** $j = 1$ **to** n
3 $y_i = y_i + a_{ij} x_j$
4 **else** $mid = \lfloor (i + i')/2 \rfloor$
5 **spawn** MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
6 MAT-VEC-MAIN-LOOP$(A, x, y, n, mid + 1, i')$
7 **sync**

A: input matrix.
x: input vector.
y: output vector.
n: the dimension of the vector.
i and i': start and end positions.



Analysis

- Recurrence for the span.
  - $T_\infty(n)=\begin{cases} \Theta(n), & \text{if } n=1 \\ \max(T_\infty(n/2), T_\infty(n/2)) + c = T_\infty(n/2) + c, & \text{if } n>1 \end{cases}$
  - We need to solve the recurrence.

    $T_\infty(n/2) = T_\infty(n/4) + c$

    $T_\infty(n/4) = T_\infty(n/8) + c$

    - $T_\infty(n)= T_\infty(n/2) + c$
    - $= T_\infty(n/2^2) + 2c$
    - $= T_\infty(n/2^3) + 3c$
    - $= T_\infty(n/2^i) + i*c$
    - In order to get to the base case, let $2^i=n$, so $i=\lg n$.
    - $T_\infty(n)= T_\infty(n/2^i) + i*c$
    - $= T_\infty(1) + \lg n*c$
    - $= \Theta(n) + \Theta(\lg n)$
    - $= \Theta(n)$
  - So the span is $\Theta(n)$.

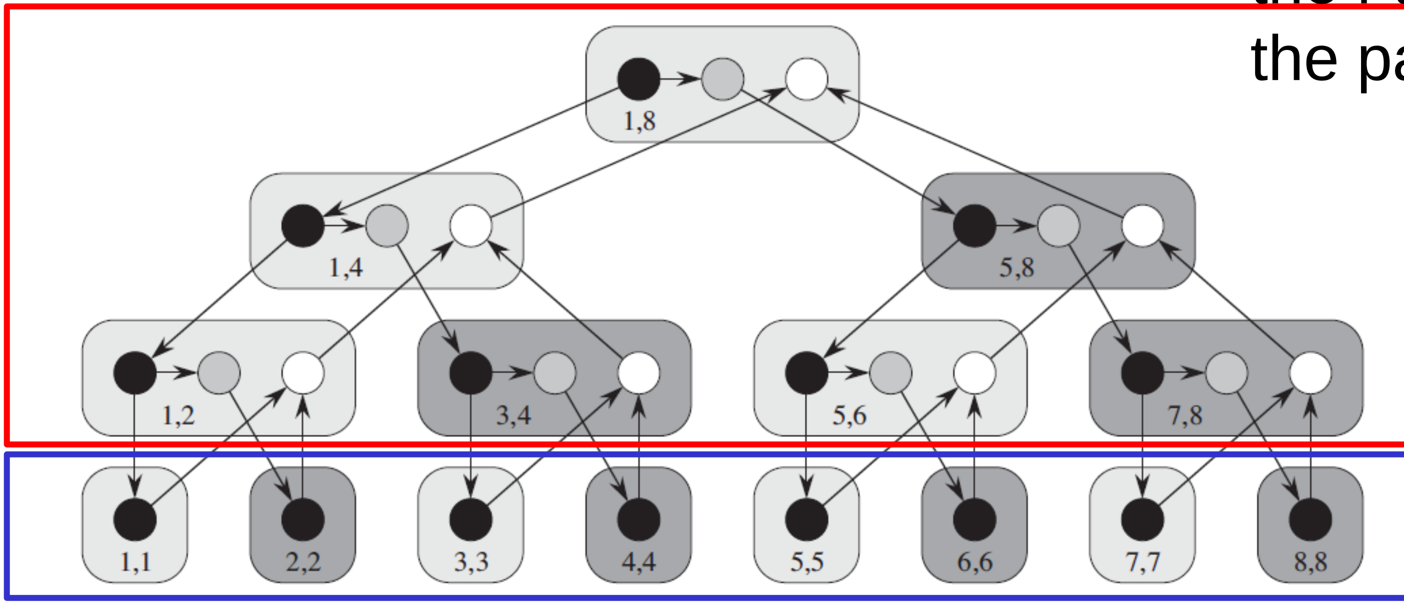
Analysis

- Recurrence for the work.
  - $T_1(n)=\begin{cases}\Theta(n), & \text{if } n=1\\ 2T_1(n/2)+c, & \text{if } n>1\end{cases}$
  - The work is $\Theta(n^2)$ due to the nested loops in 5-7
- The parallelism $\Theta(n^2)/\ \Theta(n)=\ \Theta(n)$

Analysis of parallel loops

iteration$_\infty$(*i*): the span of the i-th iteration in the parallel loop.

- $T_\infty(n) = \Theta(\lg n) + max_{1 \le i \le n}$ iteration$_\infty$(*i*)



Height of the recursion tree is $\Theta(\lg n)$

$max_{1 \le i \le n}$ iteration$_\infty$(*i*)

- Arraycopy
  - $max_{1 \le i \le n}$ iteration$_\infty$(*i*) = $\Theta(1)$, constant time, because of no iteration.
  - $T_\infty(n)=\Theta(\lg n) + \Theta(1) = \Theta(\lg n)$
- Matrix-vector multiplication
  - $max_{1 \le i \le n}$ iteration$_\infty$(*i*) = $\Theta(n)$
  - $T_\infty(n)= \Theta(\lg n) + \Theta(n) = \Theta(n)$

Agenda

- Background
- Nested parallelism
- Work, span, and parallelism
- Parallel loops
- Multithreaded merge sort

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

$T(n) = aT(n/b) + f(n)$

$a=2,\ b=2,\ f(n) = n$

$\rightarrow n^{\log_b a} = n^{\log_2 2} = n == f(n)$

$T(n) = \Theta(n\lg n)$

- Run-time?
  - $T(n)=2T(n/2)+\Theta(n)$
  - $\Theta(n\lg n)$

Multithreaded merge sort

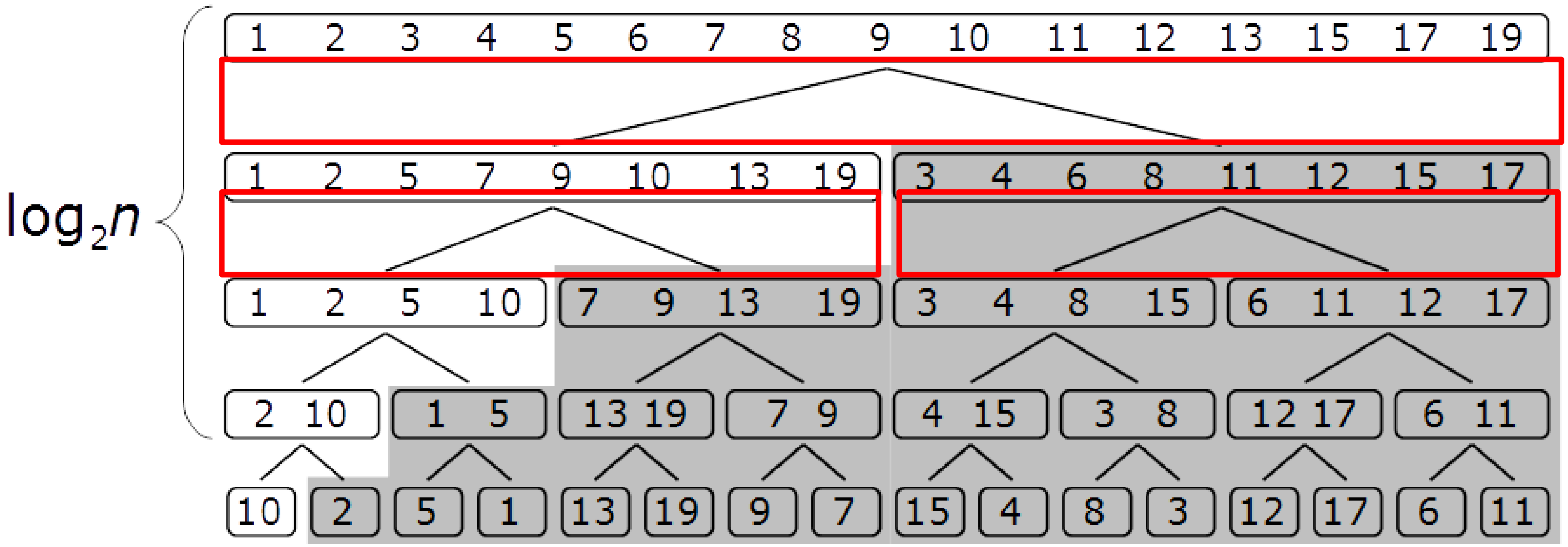
```
Merge-Sort'(A, p, r)
if p < r then
    q = ⌊(p+r)/2⌋
    spawn Merge-Sort'(A, p, q)
    Merge-Sort'(A, q+1, r)
    sync
    Merge(A, p, q, r)
```

- Work: use W to denote T_1
  - $W(n)=2W(n/2)+ \Theta(n)$
  - $=\Theta(n\lg n)$, the same with serialized merge sort.
- Span: use S to denote T_∞
  - $S(n)=\max(S(n/2), S(n/2))+ \Theta(n)$
  - $=S(n/2)+\Theta(n)$
  - $=\Theta(n)$
- Parallelism
  - $W(n)/S(n)= \Theta(n\lg n)/\Theta(n) = \Theta(\lg n)$.

The parallelism is rather low.
Can we do better, i.e., higher parallelism?
If yes, which part can we further improve?

Merge

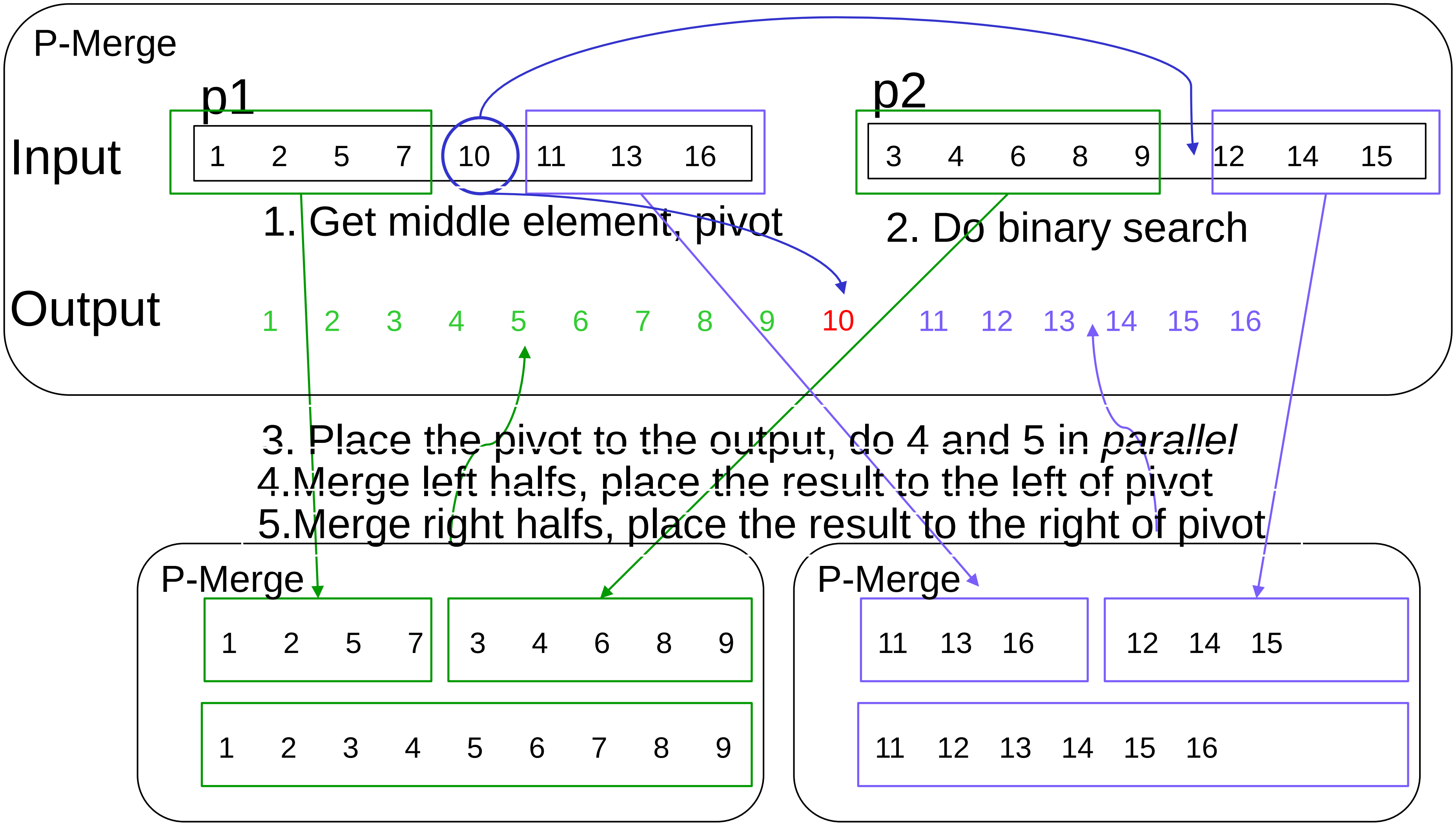
- The bottleneck: the Merge() function is very serial.



  - At the top level, only one processor does $\Theta(n)$ work in serial!
  - At the second level, only two processors do $\Theta(n)$ work.
  - …
- Can we make Merge more parallel?

Multithreaded merge

Main idea – make Merge() divide-and-conquer and use nested parallelism.



P-Merge

T: input array.
p1… r1 and p2..r2 in T: two sorted sub-arra
A: output array.
p3: starting position in A.

P-MERGE$(T, p_1, r_1, p_2, r_2, A, p_3)$

$n_1 = r_1 - p_1 + 1$
$n_2 = r_2 - p_2 + 1$
if $n_1 < n_2$ // ensure that $n_1 \geq n_2$
    exchange p_1 with p_2
    exchange r_1 with r_2
    exchange n_1 with n_2
if $n_1 == 0$ // both empty?
    return
else $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
    $q_2 = \text{BINARY-SEARCH}(T[q_1], T, p_2, r_2)$
    $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
    $A[q_3] = T[q_1]$
    spawn P-MERGE$(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3)$
    P-MERGE$(T, q_1 + 1, r_1, q_2, r_2, A, q_3 + 1)$
    sync

Why do we need to make sure $n_1 \geq n_2$?

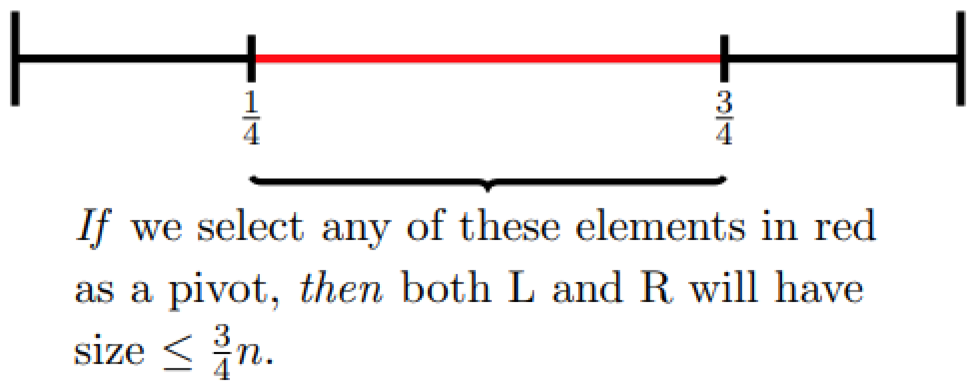
q_1: the middle element's index. $T[q_1]$: the pivot element. Binary search for $T[q_1]$ from p_2 to r_2 elements in T.

Place the pivot in the right place in A

Merge the left and right halves in parallel.

P-Merge

- Why do we need to make sure $n_1 \geq n_2$?
- It makes sure that the maximum number of elements in either of the divide-and-conquer merge calls is 3n/4
- $n_2 = (n_2 + n_2)/2 \leq (n_1 +$
- In the worst case:
  - We have $\lfloor n_1/2 \rfloor$ elem
  - We have n_2 elements
    - $\lfloor n_1/2 \rfloor + n_2$
    - $\leq n_1/2 + n_2 = n_1/2 + n_2/2 + n_2/2$
    - $= (n_1 + n_2)/2 + n_2/2$
    - $= n/2 + n_2/2$
    - $\leq n/2 + n/4$
    - $= 3n/4$



Running time for binary search.

- Span: $S(n) = \max(S(\alpha n), S((1-\alpha)n)) + \boxed{\Theta(lgn)}$
- $= S(3n/4) + \Theta(lgn)$.
- $S(n) = \Theta(lg^2 n)$

See exercise 4.6-2, CLRS. Extension to Master Method.

Multithreaded Merge Sort with P-Merge

```
P-MERGE-SORT(A, p, r, B, s)
  n = r − p + 1
  if n == 1
      B[s] = A[p]
  else let T[1 .. n] be a new array
      q = ⌊(p + r)/2⌋
      q′ = q − p + 1
      spawn P-MERGE-SORT(A, p, q, T, 1)
      P-MERGE-SORT(A, q + 1, r, T, q′ + 1)
      sync
      P-MERGE(T, 1, q′, q′ + 1, n, B, s)
```

P-Merge-Sort P-Merge

- Work: W(n)= Θ(n*lgn*)
- Span: S(n)=max(S(n/2)+S(n/2))+Θ(*lg*^{2}*n*) =S(n/2)+ Θ(*lg*^{2}*n*)
  - Θ(*lg*^{3}*n*)
- Parallelism: Θ(n *lgn*) / Θ(*lg*^{3}*n*) = Θ(n / *lg*^{2}*n*)
  - This is a better (higher) parallelism compared to Θ(*lgn*).

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)

ILO of Lecture 10

- Multithreaded algorithms
  - to understand the model of dynamic multithreading, including **nested parallelism** and **parallel loops**;
  - to understand **work, span,** and **parallelism** — the concepts necessary for the analysis of multithreaded algorithms;
  - to understand and be able to analyze the multithreaded merge sort algorithm.