Advanced Algorithms

Lecture 10
Multithreaded Algorithms

Bin Yang

byang@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 processing "cores".
    - My laptop: Intel® Core™ i7-6600U Processor has 2 cores.
  - Clusters / Intermediate price.
    - Individual computers that are connected with a dedicated network.
  - Supercomputers / High price.
    - Combination of custom architectures and custom networks to deliver the highest performance.

- Memory models
  - Shared memory: all processors can access any location of 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?

$F(1)=1$

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 …

$\text{FIB}(n)$

$\quad$ 1 $\quad$ **if** $n \leq 1$
$\quad$ 2 $\qquad$ **return** n
$\quad$ 3 $\quad$ **else** $x = \text{FIB}(n-1)$
$\quad$ 4 $\qquad\quad$ $y = \text{FIB}(n-2)$
$\quad$ 5 $\qquad$ **return** $x + y$

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

Fibonacci Numbers (3)

F(6) = 8
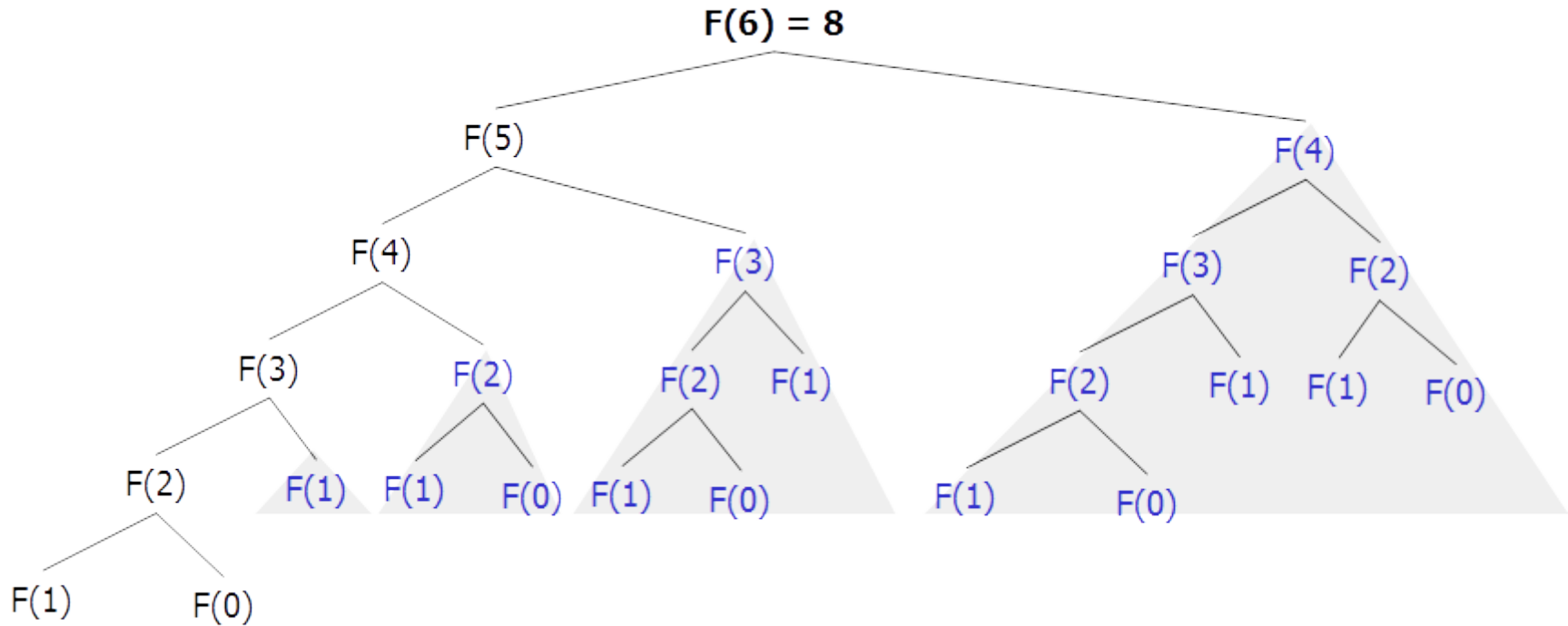
- We keep calculating the same value over and over!
  - Sub-problems are overlapping – they share sub-sub-problems.
- Do you still remember how we can avoid calculating the same sup-problems?

Fibonacci Numbers (4)

- Recurrence
  - $T(n) = T(n-1) + T(n-2) + \Theta(1)$
- 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.
  - $T(n) = 2T(n-2) + a$
  - $T(n-2) = 2T(n-4) + a$
  - $T(n) = 2^2 T(n-4) + (2+1)a$
  - $T(n-4) = 2T(n-6) + a$
  - $T(n) = 2^3 T(n-6) + (2^2+2+1)a$
  - $T(n) = 2^i T(n-2*i) + (2^{i-1} + \ldots + 2 + 1)a = 2^i T(n-2*i) + a \sum_{k=0}^{i-1} 2^k$
  - 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).

F(6) = 8

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)

1 **if** $n \leq 1$
2   **return** n
3 **else** $x = $ **spawn** P-FIB$(n-1)$
4        $y = $ P-FIB$(n-2)$
5   **sync**
6   **return** $x + y$

FIB(n)

1 **if** $n \leq 1$
2   **return** n
3 **else** $x = $ FIB$(n-1)$
4        $y = $ FIB$(n-2)$
5   **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***.

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** $n \le 1$
2   **return** n
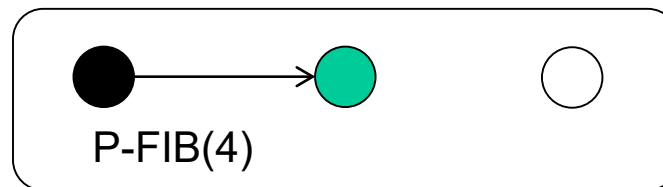3   **else** $x = $ **spawn** P-FIB$(n-1)$
4       $y = $ P-FIB$(n-2)$
5   **sync**
6   **return** $x + y$
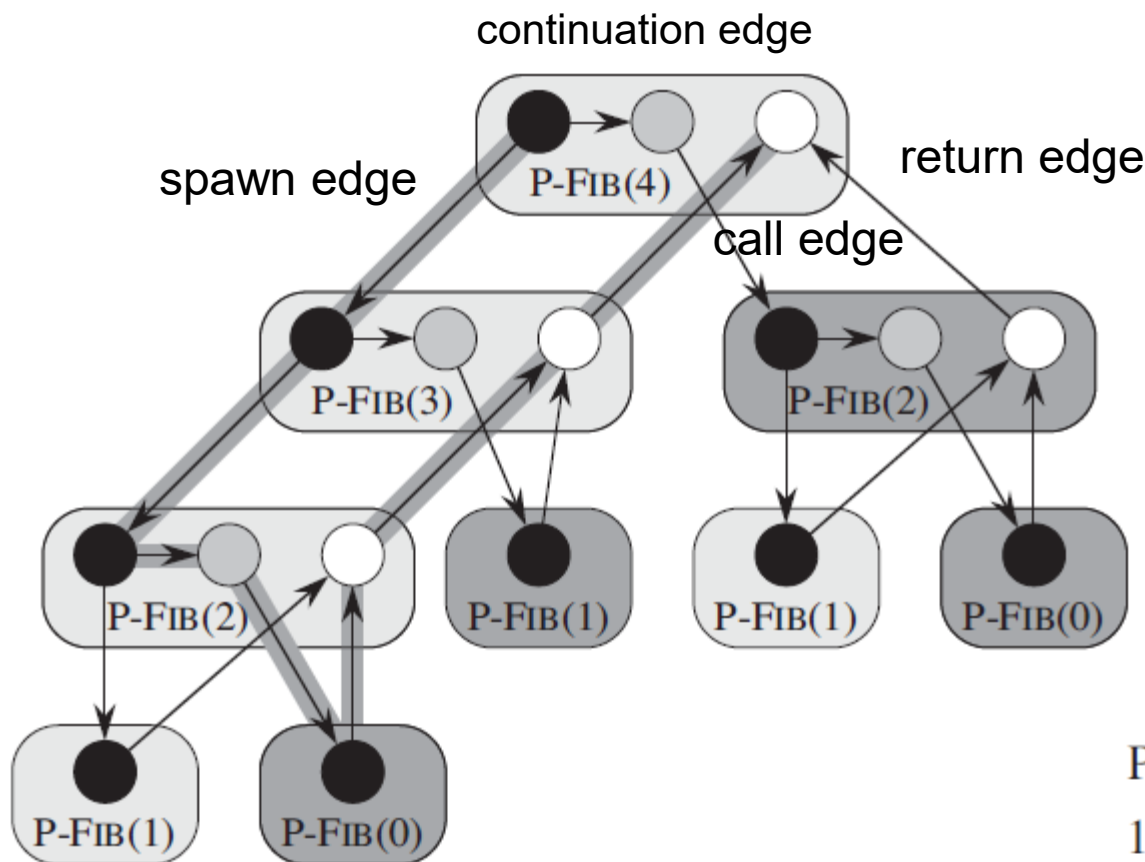
● Lines 1-3
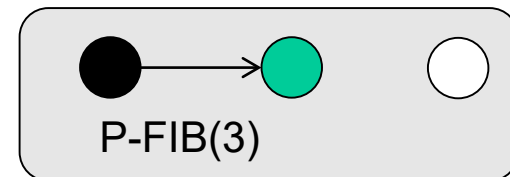
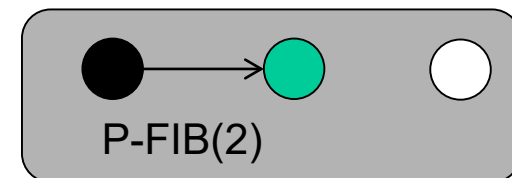● Lines 4-5

○ Line 6

P-FIB(4)

Computation DAG

continuation edge

spawn edge

return edge

call edge

P-FIB(4)

P-FIB(3)

P-FIB(2)

P-FIB(2)

P-FIB(1)

P-FIB(1)

P-FIB(0)

P-FIB(2)

P-FIB(1)

P-FIB(0)

Spawned procedure

P-FIB(3)

Called procedure

P-FIB(2)

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

P-FIB(n)

1   **if** $n \le 1$
2   **return** n
3   **else** $x =$ **spawn** P-FIB$(n - 1)$
4       $y =$ P-FIB$(n - 2)$
5   **sync**
6   **return** $x + y$

17

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.

Speedup

- Speedup: T_1/T_P:
    - How many times faster the computation on P processors than on a single processor.
    - Recall the work law says that $T_P \geq T_1 / P$, 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 $T_\infty = 8$

- 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 $T_\infty = 8$

- 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: $\text{Parallelism}/P = T_1 / PT_\infty$
  - 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

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:

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

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

- If they are joined in parallel:
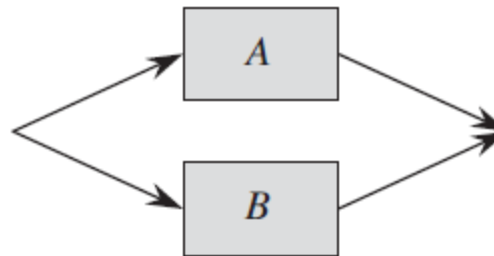
Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

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

Analyzing multithreaded Fibonacci

- $T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + c = T_\infty(n-1) + c$
- $T_\infty(n) = \Theta(n)$.

- 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)
01 parallel for i = 1 to sizeof(A) do
02    B[i] = A[i]
```

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

```
ArrayCopyRecursive(A, B, l, r)
01 if l > r then return
02 if l = r then B[l] = A[r]
03 else
04    q = ⌈(l+r)/2⌉
05    spawn ArrayCopyRecursive(A, B, l, q-1)
06            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)
01 if l > r then return
02 if l = r then B[l] = A[r]
03 else
04    q = ⌈(l+r)/2⌉
05    spawn ArrayCopyRecursive(A, B, l, q-1)
06           ArrayCopyRecursive(A, B, q, r)
```

- Using one spaw
  - Span: $T_\infty(n) = \max(T_\infty(n/2), T_\infty(n/2))$ **+ c** $= T_\infty(n/2)$ **+ c** $= \Theta(\lg n)$
  - Work: $T_1(n) = 2T_1(n/2)$ **+ c** $= \Theta(n)$
- Using two spaws
  - Span: $T_\infty(n) = \max(T_\infty(n/3), T_\infty(n/3), T_\infty(n/3))$ **+ 2c** $= T_\infty(n/3)$ **+ 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: $T_\infty(n) = \max(T_\infty(n/n), ..., T_\infty(n/n)) +$ **(n-1)*c**$= a+$**(n-1)*c**$ = \Theta(n)$
  - Work: $T_1(n) = nT_1(1) + (n-1)*c = \Theta(n)$

- Assume that we have P processing units, and we make P-1 spawn threads.
  - Span: $T_P(n) = \max(T_P(n/P), ..., T_P(n/P)) +$ **(P-1)*c**$= T_P(n/P)+$**(P-1)*c**$ = \Theta(Plg_P n)$
  - Assume that P=2, $T_P(n) = \Theta(2lg_2 n) = \Theta(lgn)$
  - Assume that P=3, $T_P(n) = \Theta(3lg_3 n) = \Theta(lgn)$
  - Assume that P=n, $T_P(n) = \Theta(nlg_n n) = \Theta(n)$

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( )
1   $x = 0$
2   **parallel for** $i = 1$ **to** 2
3        $x = x + 1$
4   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.

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

$a11$	$a12$	$a13$	$a14$
$a21$	$a22$	$a23$	$a24$
$a31$	$a32$	$a33$	$a34$
$a41$	$a42$	$a43$	$a44$

$x1$
$x2$
$x3$
$x4$

MAT-VEC(A, x)

1 $n = A.rows$
2 let y be a new vector of length n
3 **parallel for** $i = 1$ **to** n
4      $y_i = 0$
5 **parallel for** $i = 1$ **to** n
6   **for** $j = 1$ **to** n
7          $y_i = y_i + a_{ij} x_j$
8 **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
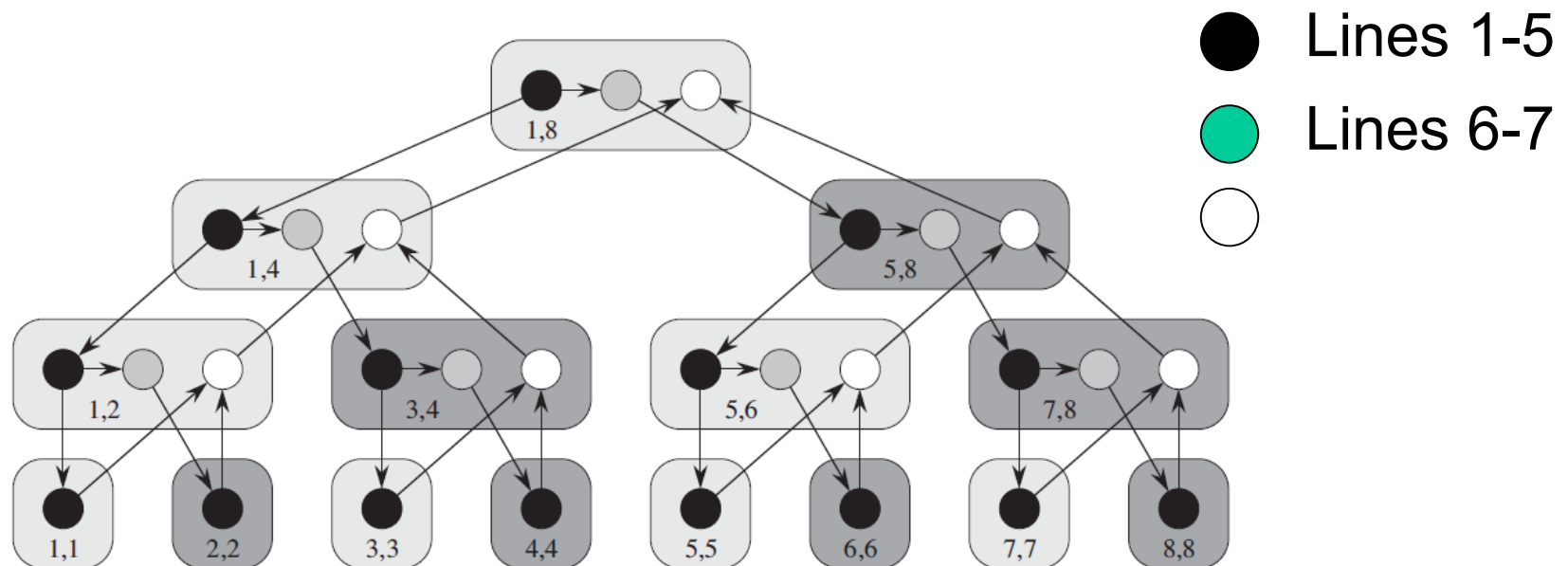7 $y_i = y_i + a_{ij}x_j$

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.

● Lines 1-5

● Lines 6-7

○

Analysis

- Recurrence for the span.
    - $T_\infty(n)=$
        - $\Theta(n)$,   if n=1
        - $Max(T_\infty(n/2), T_\infty(n/2)) + c = T_\infty(n/2)+c$,   if n>1
    - We need to solve the recurrence.
        - $T_\infty(n)= T_\infty(n/2)+c= T_\infty(n/2^2)+2c = T_\infty(n/2^3)+3c$
        - $=\ldots= T_\infty(n/2^i)+i*c$
        - In order to get to the base case, let $2^i=n$, so i=lgn.
        - $T_\infty(n)= T_\infty(n/2^i)+i*c =T_\infty(1)+clgn= \Theta(n)+ \Theta(lgn)= \Theta(n)$
    - So the span is $\Theta(n)$.

- Recurrence for the work.
    - $T_1(n)=$
        - $\Theta(n)$,   if n=1
        - $2T_1(n/2) + c$,   if n>1
    - The work is $\Theta(n^2)$

- The parallelism $\Theta(n^2)/ \Theta(n)= \Theta(n)$

Analysis of parallel loops

- $T_\infty(n) = \Theta(\lg n) + max_{1 \le i \le n}$ iteration$_\infty$(*i*)
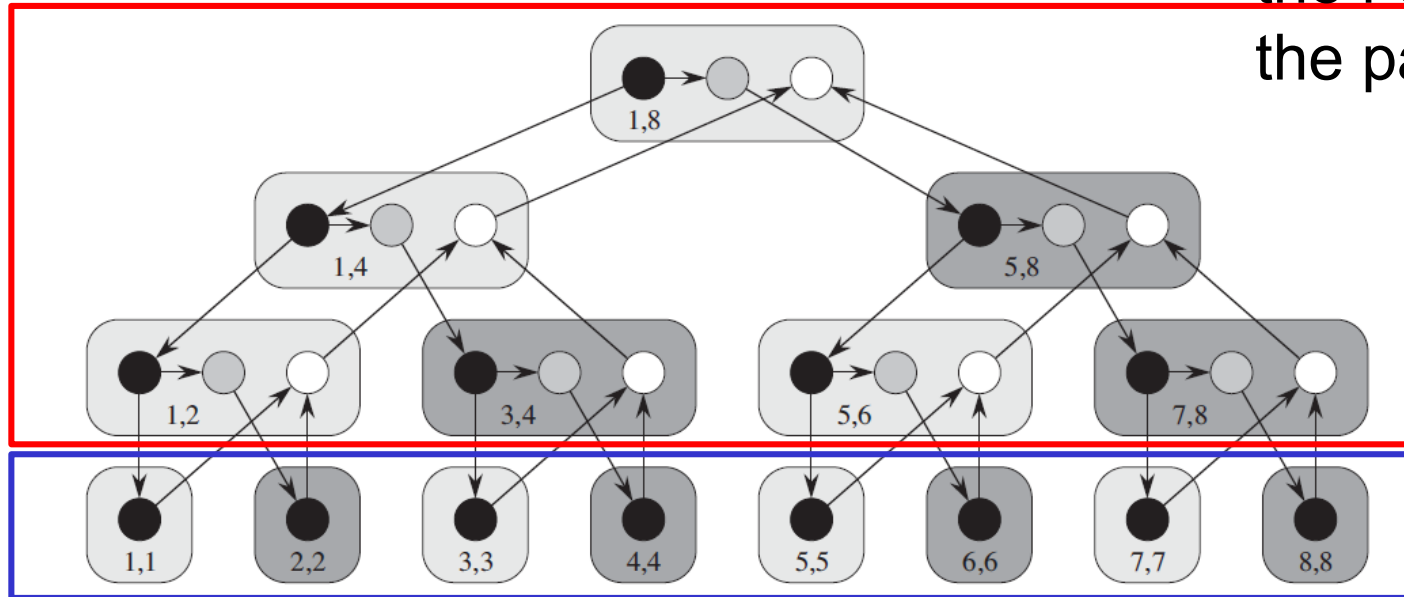
Height of the recursion tree is $\Theta(\lg n)$

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

- Arraycopy

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

- Matrix-vector multiplication

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

34

Agenda

- Background

- Nested parallelism

- Work, span, and parallelism

- Parallel loops

- Multithreaded merge sort

Merge sort

```
Merge-Sort(A, p, r)
01 if p < r then
02     q = ⌊(p+r)/2⌋          } Divide
03     Merge-Sort(A, p, q)
04     Merge-Sort(A, q+1, r)  } Conquer
05     Merge(A, p, q, r)      } Combine
```

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

Multithreaded merge sort

```
Merge-Sort'(A, p, r)
01 if p < r then
02     q = ⌊(p+r)/2⌋
03     spawn Merge-Sort'(A, p, q)
04     Merge-Sort'(A, q+1, r)
05     sync
06     Merge(A, p, q, r)
```
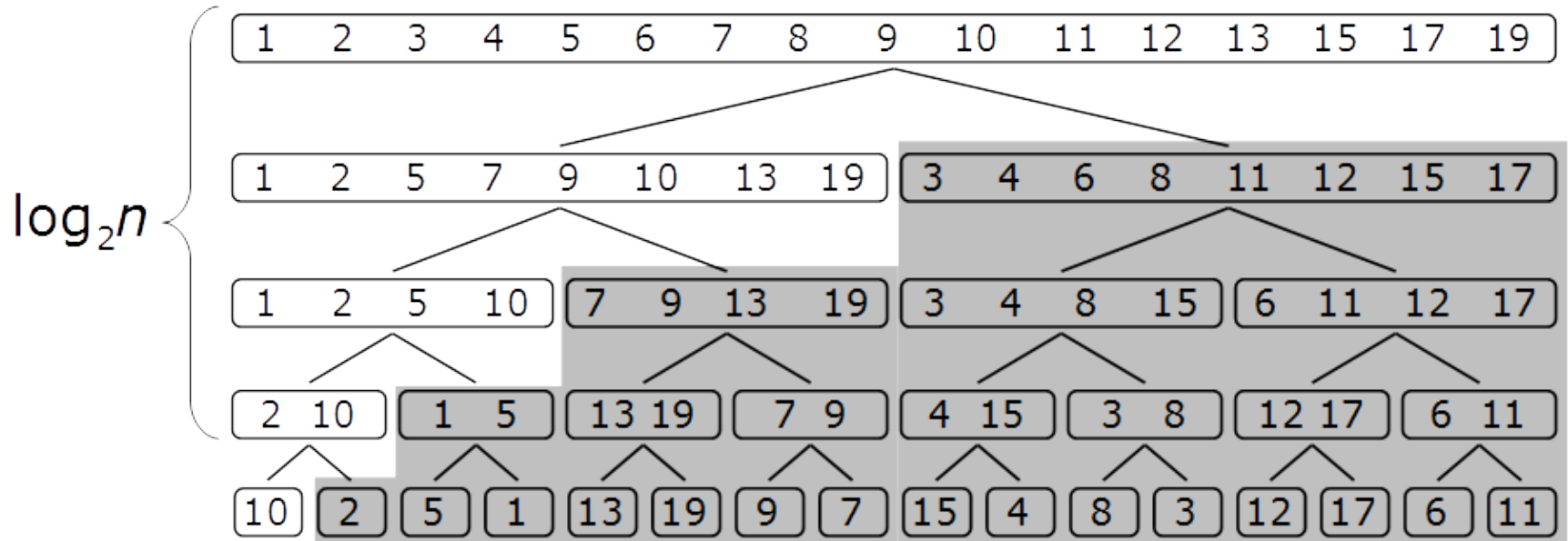
- Work: use W to denote T_1
  - $W(n)=2W(n/2)+ \Theta(n)$
  - $\Theta(nlgn)$, the same with serialized merge sort.

- 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
  - $W(n)/S(n)= \Theta(nlgn)/\Theta(n) = \Theta(lgn)$.

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

Merge

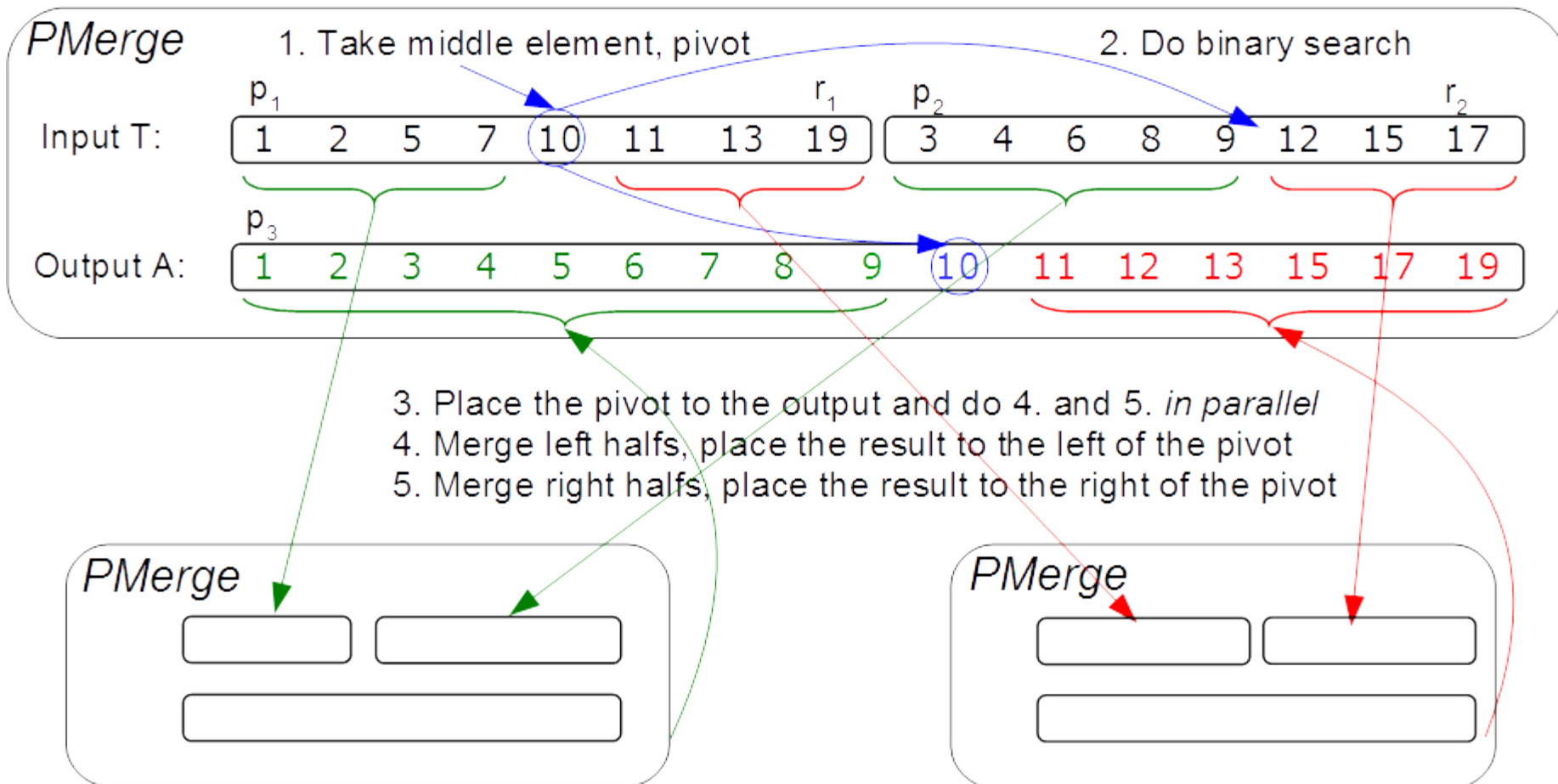- The bottleneck: the Merge() function is very serial.

- 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[y]
A: output array.
p3: starting position in A.

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

```
1   n₁ = r₁ − p₁ + 1
2   n₂ = r₂ − p₂ + 1
3   if n₁ < n₂                        // ensure that n₁ ≥ n₂
4       exchange p₁ with p₂
5       exchange r₁ with r₂
6       exchange n₁ with n₂
7   if n₁ == 0                        // both empty?
8       return
9   else q₁ = ⌊(p₁ + r₁)/2⌋
10      q₂ = BINARY-SEARCH(T[q₁], T, p₂, r₂)
11      q₃ = p₃ + (q₁ − p₁) + (q₂ − p₂)
12      A[q₃] = T[q₁]
13      spawn P-MERGE(T, p₁, q₁ − 1, p₂, q₂ − 1, A, p₃)
14      P-MERGE(T, q₁ + 1, r₁, q₂, r₂, A, q₃ + 1)
15      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.

40

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.
- $n_2 = (n_2 + n_2)/2 \leq (n_1 + n_2)/2 = n/2$
- In the worst case:
  - We have $\lfloor n_1/2 \rfloor$ elements from $T[p_1 \ldots r_1]$.
  - We have n_2 elements, i.e., all elements, from $T[p_2 \ldots r_2]$.
    - $\lfloor n_1/2 \rfloor + n_2$
    - $\leq n_1/2 + n_2 = n_1/2 + n_2/2 + n_2/2$
    - $= (n_1 + n_2)/2 + n_2/2$
    - $= n/2 + n_2/2$
    - $\leq n/2 + n/4$
    - $= 3n/4$

Running time for binary search.

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

Multithreaded Merge Sort with P-Merge

P-MERGE-SORT(A, p, r, B, s)

1   $n = r - p + 1$
2   **if** $n == 1$
3         $B[s] = A[p]$
4   **else** let $T[1 .. n]$ be a new array
5         $q = \lfloor (p + r)/2 \rfloor$
6         $q' = q - p + 1$
7   **spawn** P-MERGE-SORT$(A, p, q, T, 1)$
8   P-MERGE-SORT$(A, q + 1, r, T, q' + 1)$
9   **sync**
10   P-MERGE$(T, 1, q', q' + 1, n, B, s)$

- Work: $W(n) = \Theta(n \lg n)$
- Span: $S(n) = \max(S(n/2) + S(n/2)) + \Theta(\lg^2 n) = S(n/2) + \Theta(\lg^2 n)$
  - $\Theta(\lg^3 n)$
- Parallelism: $\Theta(n \lg n) / \Theta(\lg^3 n) = \Theta(n / \lg^2 n)$
  - This is a better (higher) parallelism compared to $\Theta(\lg n)$.

To summarize

- Work: $\Theta(nlgn)$

	Merge Procedure	Span	Parallelism
Naïve merge	$\Theta(n)$	$\Theta(n)$	$\Theta(lgn)$
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.