Algorithm Analysis and Data Structures

CSCI 7432 - Fall 2022

Multithreaded Algorithms

Dr. Yao XU

Assistant Professor

Department of Computer Science

Georgia Southern University

Email: yxu@georgiasouthern.edu

Table of Contents

Dynamic Multithreading

Parallel Algorithms

- All algorithms we've discussed are ***serial algorithms***
  - Run on a uniprocessor computer
  - Execute one instruction at a time
- We will talk about ***parallel algorithms***
  - Run on a multiprocessor computer
  - Permits multiple instructions to execute concurrently
  - Explore an elegant model: ***dynamic multithreaded algorithms***

Static Threading v.s. Dynamic Multithreading

- ***Static threading***
  - Provides the programmer with an abstraction of virtual processors that are managed explicitly.
  - The programmer must specify in advance how many processors to use at each point.
- ***Dynamic multithreading***
  - Programmers specify opportunities for parallelism;
  - A concurrency platform manages the decisions of mapping these opportunities to actual static threads. – ***Scheduling problem*** (Will not be discussed)

Dynamic Multithreading

- Functionality of dynamic-multithreading supports two features:
  - ***Nested parallelism*** - allows a subroutine to be "spawned" to allow the caller to proceed while the spawned subroutine is computing its result.
  - ***Parallel loops*** - like an ordinary loop, except that the iterations of the loop can execute concurrently.
- Three "concurrency" keywords in pseudocode:
  - ***parallel***: add to loop to indicate iterations can be executed in parallel.
  - ***spawn***: create a parallel process and keep executing the current one.
  - ***sync***: wait until all active parallel threads finish.
- ***Serialization*** of a multithreaded algorithm: deleting these keywords, we get the serial algorithm for the same problem.

Example: Parallel Fibonacci (1/2)

- *Recall:* Fibonacci numbers
  - $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$
- A recursive, serial algorithm to compute F_n:
- Time complexity: $T(n) = T(n-1) + T(n-2) + \Theta(1)$
  - $T(n) \in \Theta(F_n)^*$, F_n grows exponentially in n
  - $T(n) \in \Theta(\phi^n)$, where $\phi = (1+\sqrt{5})/2$.*

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

- **Observation:** The recursive calls operate independently of each other.
  - The two recursive calls can run in parallel.
- **Q:** What improvement can we get?

**See p.775-776 of the textbook for proof of the time complexity.*

Example: Parallel Fibonacci (2/2)

- Augment pseudocode to indicate parallelism by adding the concurrency keywords **spawn** and **sync**.
- Notice that without the concurrency keywords, P-FIB(n) is identical to the algorithm FIB(n).

```
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
```

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

- ***Nested parallelism:***
  - **spawn** precedes a procedure call, as in line 3.
  - The child computes P-FIB(n - 1)
  - The parent computes P-FIB(n - 2) in parallel with the child
  - **sync** indicates parent must wait for all children to complete before continuing

Analyzing Multithreaded Algorithms

A Model for Multithreaded Execution

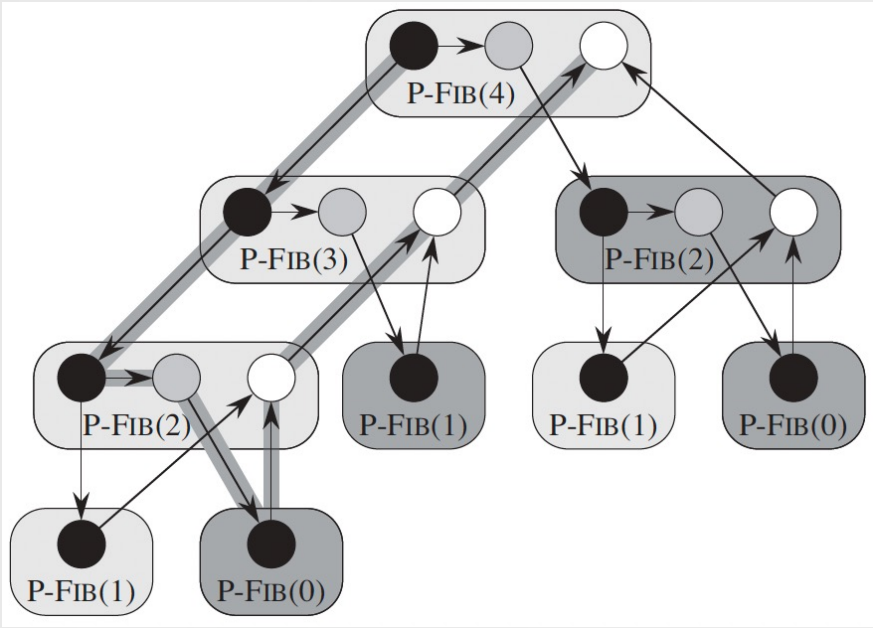
A multithreaded computation can be modeled as a ***computation DAG*** $G = (V, E)$.

- $v \in V$ represents a ***strand***: a sequence of non-parallel instructions.
- $e \in E$ represents dependencies between two strands: $(u, v) \in E$ means u must execute before v.
- If there is a directed path from u to v, then they are ***(logically) in series***; otherwise, they are ***(logically) in parallel***.

Example: Computation DAG of P-FIB(4).

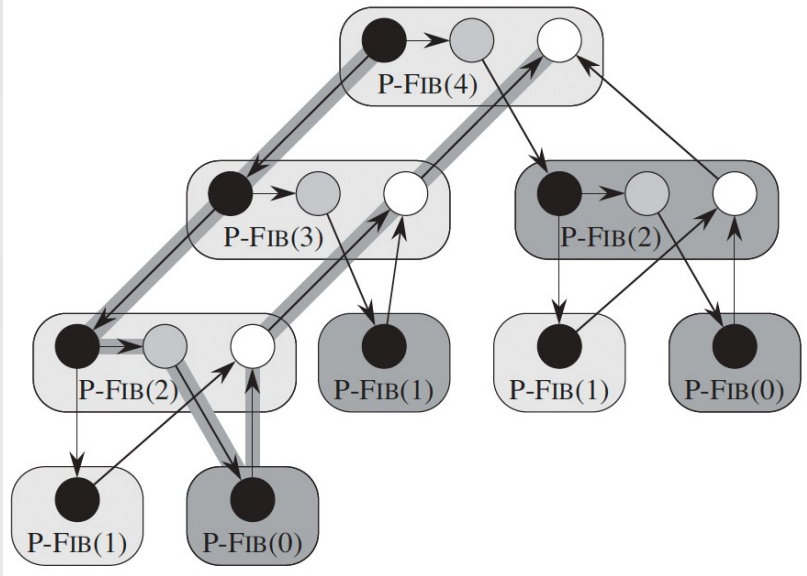
- Black nodes: lines 1-3; Grey nodes: line 4; White nodes: line 6

```
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
```



Performance Measures (1/2)

- Denote T_P = running time of an algorithm on P processors.
- Two metrics to measure time complexity of a multithreaded algorithm:
  - ***Work:*** T_1 = total time to execute the entire computation on one processor.
    - ***Work law***: $T_P \geq T_1/P$
  - ***Span:*** T_∞ = longest time to execute the strands along any path in the DAG.
    - ***Span law***: $T_P \geq T_\infty$
- **Example:** P-FIB(4)
  - Assuming 1 unit of work per strand.
  - Work: $T_1 = 17$
  - Span: $T_\infty = 8$



Performance Measures (2/2)

- ***Speedup*** of a computation on P processors: T_1/T_P $(\leq P)$
  - ***Linear speedup***: $T_1/T_P \in \Theta(P)$
  - ***Perfect linear speedup***: $T_1/T_P = P$
- The ***parallelism*** of a multithreaded computation: T_1/T_∞
  - *Ratio:* average amount of work that can be performed for each strand along the longest path in the computation DAG.
  - *Upper Bound:* maximum possible speedup that can be achieved on any number of processors.
  - *Limit:* limit on the possibility of attaining perfect linear speedup.
- **Example:** P-FIB(4)
  - The parallelism is $\frac{T_1}{T_\infty} = \frac{17}{8}$ (achieving $\gg 2$ speedup is impossible)

Analysis of P-FIB(n)

- Analyzing work: same as FIB(n)

$$T_1(n) = T_1(n-1) + T_1(n-2) + \Theta(1),$$
$$\Rightarrow T_1(n) \in \Theta(\phi^n).$$

- Analyzing span:

$$T_\infty(n) = \max\{T_\infty(n-1), T_\infty(n-2)\} + \Theta(1),$$
$$\Rightarrow T_\infty(n) \in \Theta(n).$$

- The parallelism of P-FIB(n): $T_1/T_\infty \in \Theta(\phi^n/n)$.
  - It grows dramatically as n gets large.
  - There is potential for near perfect linear speedup.

```
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
```

Parallel Loops Example (1/4)

Multiply an $n \times n$ matrix $A = (a_{ij})$ by an n-vector $x = (x_j)$.

- Result is an n-vector $y = (y_i)$, where $y_i = \sum_{j=1}^{n} a_{ij} x_j$, for $i = 1, 2, \cdots, n$.

Example: $A = \begin{bmatrix} 1 & 2 & 1 & 3 \\ 0 & 3 & 2 & 1 \\ 2 & 1 & 1 & 0 \\ 1 & 0 & 2 & 1 \end{bmatrix}$ and $x = \begin{bmatrix} 2 \\ 1 \\ 0 \\ 2 \end{bmatrix}$

$$\Rightarrow y = Ax = \begin{bmatrix} 10 \\ 5 \\ 5 \\ 4 \end{bmatrix}$$

- A parallel algorithm MAT-VEC(A, x):

```
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
```

Parallel Loops Example (2/4)

- A compiler can implement a **parallel for** loop via divide-and-conquer.
- Lines 5-7 can be implemented with MAT-VEC-MAIN-LOOP$(A, x, y, n, 1, n)$.

```
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
```

```
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 = ⌊(i + i')/2⌋
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
```

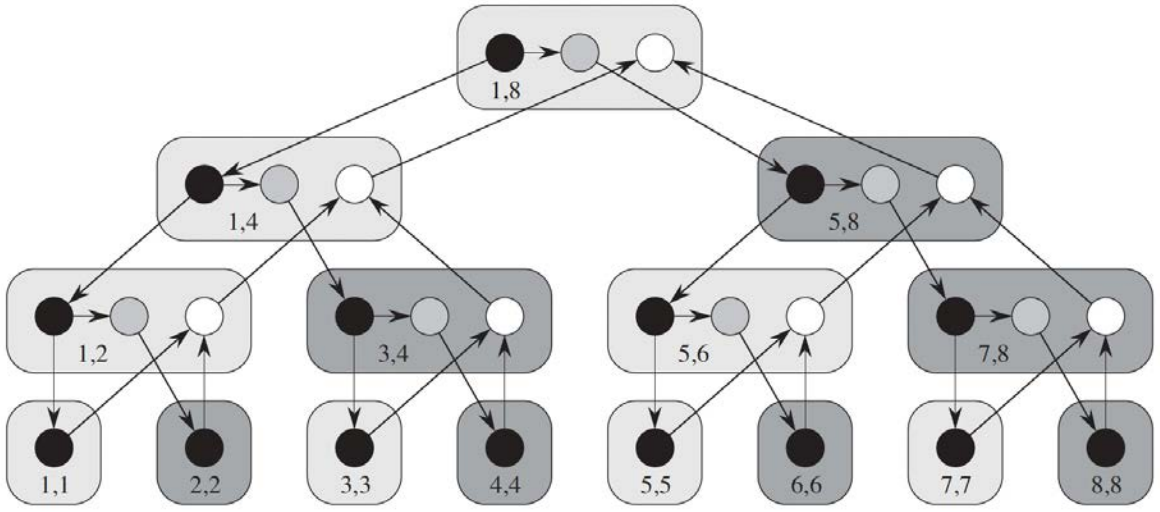
- It is not realistic to think that all n iterations in a loop can be spawned simultaneously with no extra work.

Parallel Loops Example (3/4)

- **Example:** A DAG representing the computation of MAT-VEC-MAIN-LOOP$(A, x, y, 8, 1, 8)$.
  - Black nodes: lines 1-5
  - Grey nodes: line 6
  - White nodes: line 7
  - Each leaf node corresponds to one iteration of the loop.
- To parallelize a **for** loop with n iterations, the extra work of recursive spawning is:

$$T_{spawn}(n) \in \Theta(\log n).$$

```
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 = ⌊(i + i')/2⌋
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
```



Parallel Loops Example (4/4)

- Analyzing work: $T_1(n) \in \Theta(n^2)$
- Analyzing span: $T_\infty(n) = T_\infty^1(n) + T_\infty^2(n) + \Theta(1)$,
  - parallel **for** loop in lines 3-4:

    $$T_\infty^1(n) = T_{spawn}(n) + \Theta(1) \in \Theta(\log n),$$

  - parallel **for** loop in lines 5-7:

    $$T_\infty^2(n) = T_{spawn}(n) + \Theta(n) \in \Theta(n),$$

  where $T_{spawn}(n) \in \Theta(\log n)$.

  $\Rightarrow T_\infty(n) \in \Theta(n)$.
- The parallelism is: $T_1/T_\infty \in \Theta(n)$

```
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
```

Q: Can we improve by making the inner for loop (lines 6-7) parallel as well?

A: NO.

Race Conditions

- An algorithm is ***deterministic*** if it always does the same thing on the same input; it is ***nondeterministic*** if the result might vary from run to run.
- A multithreaded algorithm that is intended to be deterministic fails to be when it contains a "***determinacy race***."
- A ***determinacy race*** occurs when two logically parallel instructions access the same memory location and at least one of them performs a write.
- **Example:** RACE-EXAMPLE should always print 2, but it could print 1 instead.

```
RACE-EXAMPLE( )
1  x = 0
2  parallel for i = 1 to 2
3       x = x + 1
4  print x
```

To cope with races, ensure that

- strands that operate in parallel are **independent**,
- including all iterations in a parallel loop.

Multithreaded Merge Sort

Parallelizing Merge Sort

Serial Merge Sort:

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- Work: $T_1(n) = 2T_1(n/2) + cn$
  $\Rightarrow T_1(n) \in \Theta(n \log n)$.
- Span: $T_\infty(n) = T_\infty(n/2) + cn$
  $\Rightarrow T_\infty(n) \in \Theta(n)$.
- Parallelism: $T_1/T_\infty \in \Theta(\log n)$

A parallel Merge Sort:

```
MERGE-SORT'(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      spawn MERGE-SORT'(A, p, q)
4      MERGE-SORT'(A, q + 1, r)
5      sync
6      MERGE(A, p, q, r)
```

- The serial MERGE procedure seems to be inherently serial.
- How to make it parallel?

The Merge Procedure

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

- **Input:** Array A with indices p, q, r, s.t.
  - $p \leq q < r$
  - $A[p..q]$ and $A[q+1..r]$ are sorted
- **Output:** $A[p..r]$ is sorted.

Example:

- $A[p..r] = \langle 1, 4, 5, 7, 9, 0, 2, 3, 6, 8 \rangle$
- $p = 1, q = 5, r = 10. n_1 = 5, n_2 = 5.$

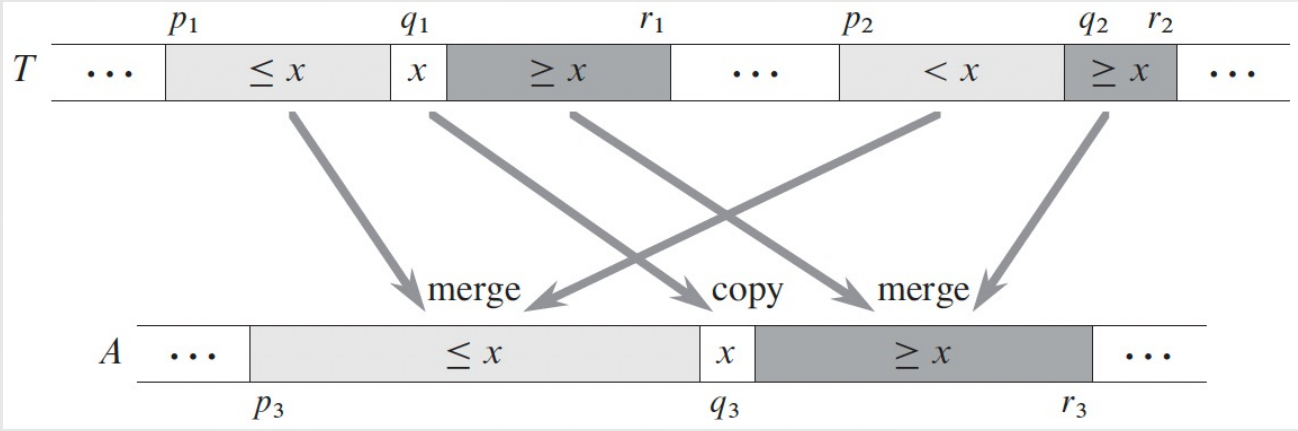
	1	2	3	4	5	6
L	1	4	5	7	9	∞

	1	2	3	4	5	6
R	0	2	3	6	8	∞

	1	2	3	4	5	6	7	8	9	10
A										

A Divide-and-Conquer Merge (1/2)

- There is a divide-and-conquer strategy to make it parallel.
- **Idea:** Break the two sorted lists into four, two merged to form the head and two merged to form the tail.

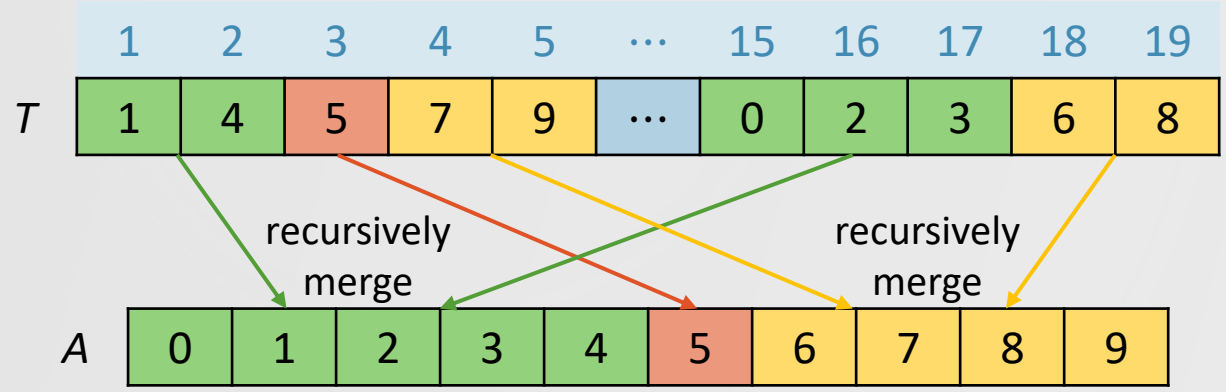


1. Choose the longer list to be the first one $T[p_1..r_1]$.
2. Find the median $x = T[q_1]$, where $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$.
3. Break the other list into two with binary search taking x as the key.
4. Recursively merge $T[p_1..q_1 - 1]$ and $T[p_2..q_2 - 1]$ to form the head of A, and merge $T[q_1 + 1..r_1]$ and $T[q_2..r_2]$ to form the tail of A.
5. Place x between them.

A Divide-and-Conquer Merge (2/2)

1. Choose the longer list to be the first one $T[p_1..r_1]$.
2. Find the median $x = T[q_1]$, where $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$.
3. Break the other list into two with binary search taking x as the key.
4. Recursively merge $T[p_1..q_1 - 1]$ and $T[p_2..q_2 - 1]$ to form the head of A, and merge $T[q_1 + 1..r_1]$ and $T[q_2..r_2]$ to form the tail of A.
5. Place x between them.

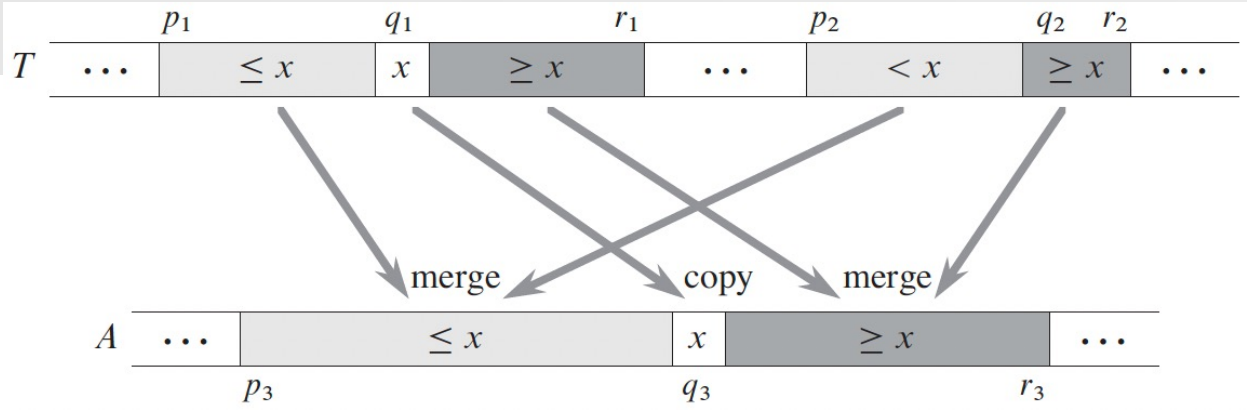
Example: $T[p_1..r_1] = \langle 1, 4, 5, 7, 9 \rangle$, $T[p_2..r_2] = \langle 0, 2, 3, 6, 8 \rangle$



The Parallel Merge Procedure

Parallel Merge:

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
 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
```




```
BINARY-SEARCH(x, T, p, r)
1  low = p
2  high = max(p, r + 1)
3  while low < high
4      mid = ⌊(low + high)/2⌋
5      if x ≤ T[mid]
6          high = mid
7      else low = mid + 1
8  return high
```

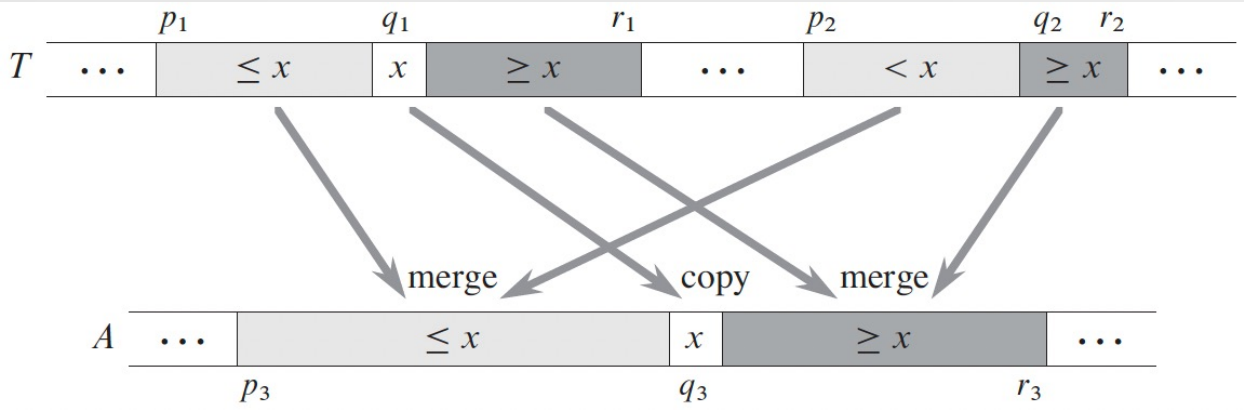
BINARY-SEARCH(x, T, p, r) returns the position of x if it were to be inserted into the list $T[p..r]$.

- It is a serial procedure.
- Running time: $O(\log n)$, where $n = r - p + 1$.

The Parallel Merge Procedure

Parallel Merge:

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
 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
```



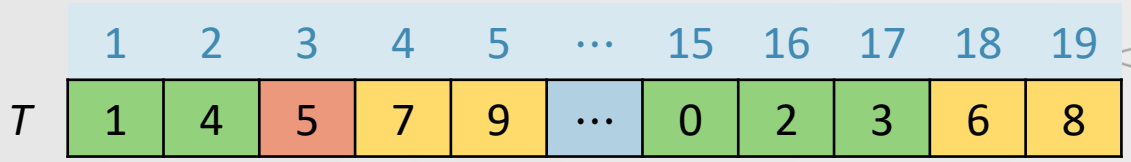

```
BINARY-SEARCH(x, T, p, r)
1  low = p
2  high = max(p, r + 1)
3  while low < high
4       mid = ⌊(low + high)/2⌋
5       if x ≤ T[mid]
6            high = mid
7       else low = mid + 1
8  return high
```

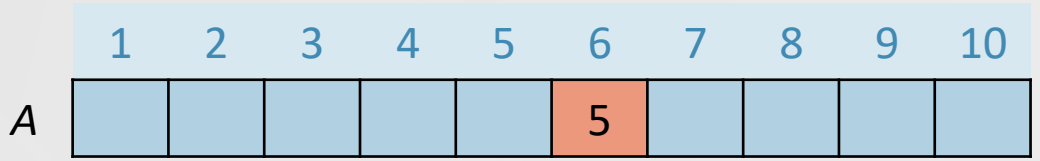
Example: $x = 5$

1	2	3	4	5
0	2	3	6	8

Example of Parallel Merge

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
 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
```



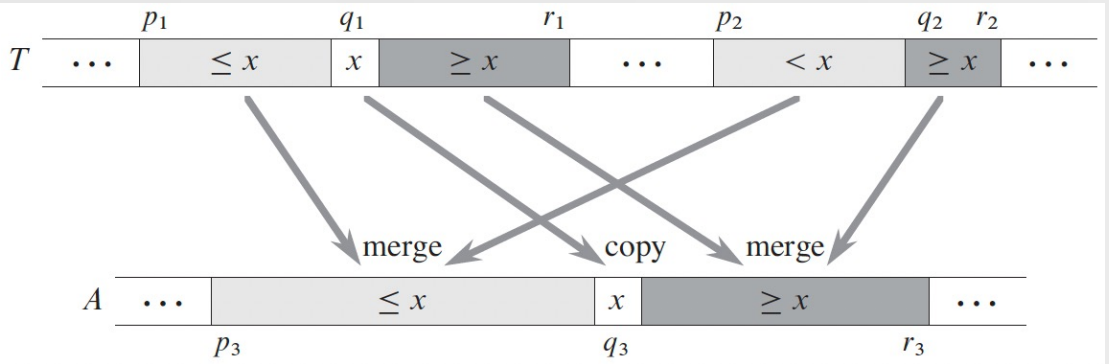


p_1	r_1	p_2	r_2	p_3	q_1	$T(q_1)$	q_2	q_3
1	5	15	19	1	3	5	18	6

Analyzing P-MERGE (1/2)

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
 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
```

Let l_1 and l_2 be the input size of the two recursive P-MERGE calls, respectively.



- $\min\{l_1, l_2\} \geq \frac{1}{2}\max\{n_1, n_2\} \geq \frac{1}{2}\left(\frac{n}{2}\right) = \frac{1}{4}n$
- Analyzing work: $PM_1(n) \in \Omega(n)$ (Why?)
  Upper bound: $PM_1(n)$
  $\leq PM_1(\alpha n) + PM_1\big((1-\alpha)n\big) + O(\log n)$,
  where $\frac{1}{4} \leq \alpha \leq \frac{3}{4}$. $\Rightarrow T_1(n) \in O(n)^*$
  Together, $PM_1(n) \in \Theta(n)$.

** See p.802 of the textbook for how $O(n)$ is obtained.*

Analyzing P-MERGE (1/2)

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
 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
```

$\min\{l_1, l_2\} \geq \frac{1}{4}n$ and $\max\{l_1, l_2\} \leq \frac{3}{4}n$.

- Analyzing work: $PM_1(n) \in \Theta(n)$
- Analyzing span:

  Worst case: $PM_\infty(n) = PM_\infty\left(\frac{3n}{4}\right) + c \log n$

  Best case: $PM_\infty(n) = PM_\infty\left(\frac{n}{4}\right) + c \log n$

  Both cases solve to $PM_\infty(n) \in \Theta(\log^2 n)$.*
- Parallelism: $PM_1/PM_\infty \in \Theta(n/\log^2 n)$

** By Case 2 of the Master Theorem.*

Analyzing P-MERGE-SORT

```
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 = ⌊(p + r)/2⌋
 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)
```

Note: For small n, we may coarsen the parallelism by using an ordinary serial sort instead.

- Analyzing work:

$$PMS_1(n) = 2PMS_1(n/2) + \Theta(n)$$

$$\Rightarrow PMS_1(n) \in \Theta(n \log n)$$

- Analyzing span:

$$PMS_\infty(n) = PMS_\infty(n/2) + \Theta(\log^2 n)$$

$$\Rightarrow PMS_\infty(n) \in \Theta(\log^3 n)^*$$

- Parallelism: $PMS_1/PMS_\infty \in \Theta(n/\log^2 n)$.
- Much better than $\Theta(\log n)$, the parallelism of MERGE-SORT' with serial Merge.

** By Case 2 of the Master Theorem.*

Thank you!
Questions?