ICS 311 #22: Multithreaded Algorithms

Outline

Readings

- CLRS 3rd Ed. Chapter 27.
- Screencasts not available

Concepts of Dynamic Multithreading

Multithreading is a crucial topic for modern computing. There are two granularities: parallel algorithms or multithreading a single algorithm, which is our emphasis here and in CLRS; and scheduling and managing multiple algorithms, each running concurrently in their own thread and possibly sharing resources, as studied in courses on operating systems and concurrent and high performance computing.

Parallel machines are getting cheaper and in fact are now ubiquitous ...

- supercomputers: custom architectures and networks
- computer clusters with dedicated networks (distributed memory)
- multi-core integrated circuit chips (shared memory)
- GPUs (graphics processing units) with multiple processors

Dynamic Multithreading

Static threading provides the programmer with an abstraction of virtual processors that are managed explicitly ("static" because they specify in advance how many processors to use at each point).

But rather than managing threads explicitly, our model is **dynamic multithreading** in which programmers specify opportunities for parallelism and a **concurrency platform** manages the decisions of mapping these to static threads (load balancing, communication, etc.).

Concurrency Constructs:

Three keywords are added, reflecting current parallel-computing practice:

- **parallel**: add to loop construct such as `for` to indicate each iteration can be executed in parallel.
- **spawn**: create a parallel subprocess, then keep executing the current process (parallel procedure call).
- **sync**: wait here until all active parallel threads created by this instance of the program finish.

These keywords specify opportunities for parallelism without affecting whether the corresponding sequential program obtained by removing them is correct. We exploit this in analysis.

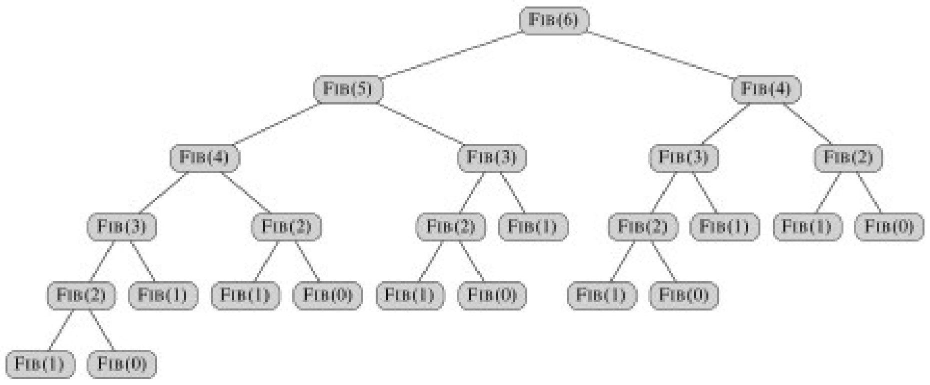
Example: Parallel Fibonacci

For illustration, we take a really slow algorithm and make it parallel. (There are much better ways to compute Fibonacci numbers; this is just for illustration.) Here is the definition of Fibonacci numbers:

$F_0 = 0.$
$F_1 = 1.$
$F_i = F_{i-1} + F_{i-2}$, for $i \geq 2$.

Here is a recursive non-parallel algorithm for computing Fibonacci numbers modeled on the above definition, along with its recursion tree:

```
FIB($n$)
if $n \leq 1$
    return $n$
else $x = $ FIB($n-1$)
    $y = $ FIB($n-2$)
    return $x + y$
```



Fib has recurrence relation $T(n) = T(n-1) + T(n-2) + \Theta(1)$, which has the solution $T(n) = \Theta(F_n)$ (see the text for substitution method proof). This grows exponentially in n, so it's not very efficient. (A straightforward iterative algorithm is much better.)

Noticing that the recursive calls operate independently of each other, let's see what improvement we can get by computing the two recursive calls in parallel. This will illustrate the concurrency keywords and also be an example for analysis:

```
P-FIB($n$)
if $n \leq 1$
    return $n$
else $x = $ spawn P-FIB($n-1$)
    $y = $ P-FIB($n-2$)
    sync
    return $x + y$
```

Notice that without the keywords it is still a valid serial program.

Logical Parallelism: The **spawn** keyword does not force parallelism: it just says that it is permissible. A scheduler will make the decision concerning allocation to processors.

However, if parallelism is used, **sync** must be respected. For safety, there is an implicit sync at the end of every procedure.

We will return to this example when we analyze multithreading.

Scheduling

Scheduling parallel computations is a complex problem: see the text for some theorems concerning the performance of a greedy centralized scheduler (i.e., one that has information on the global state of computation, but must make decisions on-line rather than in batch).

Professor Henri Casanova does research in this area, and would be an excellent person to talk to if you want to get involved.

Modeling and Measuring Dynamic Multithreading

First we need a formal model to describe parallel computations.

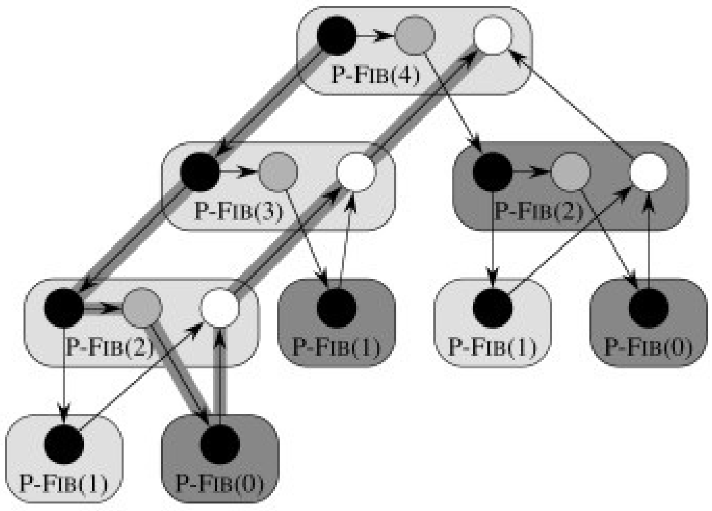
A Model of Multithreaded Execution

We will model a multithreaded computation as a **computation dag** (directed acyclic graph) $G = (V, E)$; an example for P-Fib(4) is shown:

```
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
```

Vertices in V are instructions, or **strands** = sequences of non-parallel instructions.

Edges in E represent dependencies between instructions or strands: $(u, v) \in E$ means u must execute before v.



- **Continuation Edges** (u, v) are drawn horizontally and indicate that v is the successor to u in the sequential procedure.
- **Call Edges** (u, v) point downwards, indicating that u called v as a normal subprocedure call.
- **Spawn Edges** (u, v) point downwards, indicating that u spawned v in parallel.
- **Return edges** point upwards to indicate the next strand executed after returning from a normal procedure call, or after parallel spawning at a sync point.

A strand with multiple successors means all but one of them must have spawned. A strand with multiple predecessors means they join at a sync statement.

If G has a directed path from u to v they are logically in **series**; otherwise they are logically **parallel**.

We assume an ideal parallel computer with **sequentially consistent** memory, meaning it behaves as if the instructions were executed sequentially in some full ordering consistent with orderings within each thread (i.e., consistent with the partial ordering of the computation dag).

Performance Measures

We write T_P to indicate the running time of an algorithm on P processors. Then we define these measures and laws:

Work

$\boldsymbol{T_1}$ = the total time to execute an algorithm on one processor. This is called *work* in analogy to work in physics: the total amount of computational work that gets done.

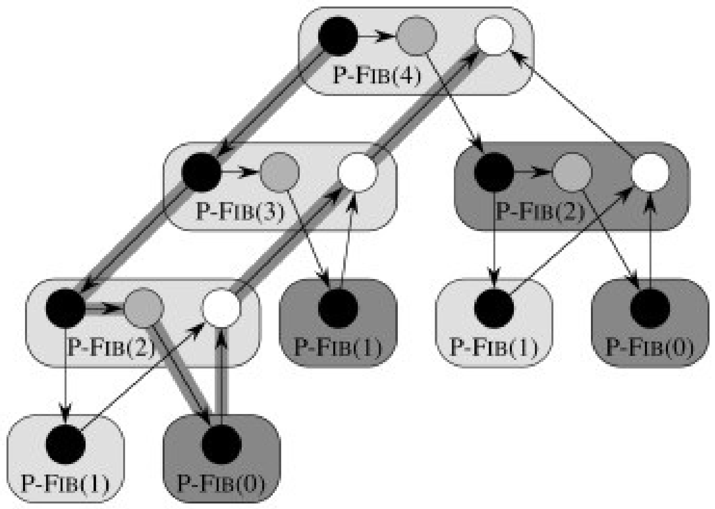
An ideal parallel computer with P processors can do at most P units of work in one time step. So, in T_P time it can do at most $P \cdot T_P$ work. Since the total work is T_1, $P \cdot T_P \geq T_1$, or dividing by P we get the **work law:**

$$T_P \geq T_1 / P$$

The work law can be read as saying that the speedup for P processors can be no better than the time with one processor divided by P (i.e., parallelism at best gives constant speedup where the constant is $1/P$,).

Span

$\boldsymbol{T_\infty}$ = the total time to execute an algorithm on an infinite number of processors (or, more practically speaking, on just as many processors as are needed to allow parallelism wherever it is possible).



T_∞ is called the *span* because it corresponds to the longest time to execute the strands along any path in the computation dag (the biggest computational span across the dag). It is the fastest we can possibly expect -- an Ω bound -- because no matter how many processors you have, the algorithm must take this long.

Hence the **span law** states that a P-processor ideal parallel computer cannot run faster than one with an infinite number of processors:

$$T_P \geq T_\infty$$

This is because at some point the span will limit the speedup possible, no matter how many processors you add.

What is the work and span of the computation dag for P-Fib shown?

Speedup

The ratio $\boldsymbol{T_1 / T_P}$ defines how much *speedup* you get with P processors as compared to one.

By the work law, $T_P \geq T_1 / P$, so $T_1 / T_P \leq P$: one cannot have any more speedup than the number of processors.

This is important: ***parallelism provides only constant time improvements*** (the constant being the number of processors) to any algorithm! ***Parallelism cannot move an algorithm from a higher to lower complexity class (e.g., exponential to polynomial, or quadratic to linear).*** Parallelism is not a silver bullet: good algorithm design and analysis is still needed.

When the speedup $T_1 / T_P = \Theta(P)$ we have **linear speedup**, and when $T_1 / T_P = P$ we have **perfect linear speedup**.

Parallelism

The ratio T_1 / T_∞ of the work to the span gives the potential parallelism of the computation. It can be interpreted in three ways:

- *Ratio* : The average amount of work that can be performed for each step of parallel execution time.
- *Upper Bound* : the maximum possible speedup that can be achieved on any number of processors.
- *Limit*: The limit on the possibility of attaining perfect linear speedup. Once the number of processors exceeds the parallelism, the computation cannot possibly achieve perfect linear speedup. The more processors we use beyond parallelism, the less perfect the speedup.

This latter point leads to the concept of **parallel slackness**,

$$(T_1 / T_\infty) / P \quad = \quad T_1 / (P \cdot T_\infty),$$

the factor by which the parallelism of the computation exceeds the number of processors in the machine. If slackness is less than 1 then perfect linear speedup is not possible: you have more processors than you can make use of. If slackness is greater than 1, then the work per processor is the limiting constraint and a scheduler can strive for linear speedup by distributing the work across more processors.

*What is the parallelism of the computation dag for P-Fib shown previously? What are the prospects for speedup at *this* n? What happens to work and span as n grows?*

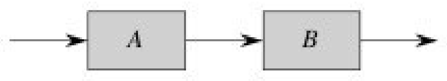
Analysis of Multithreaded Algorithms

Analyzing *work* is simple: ignore the parallel constructs and analyze the serial algorithm. For example, the work of P-Fib(n) is $T_1(n) = T(n) = \Theta(F_n)$. Analyzing *span* requires more work.

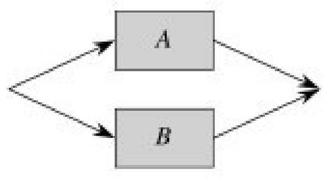
Analyzing Span

If in series, the span is the sum of the spans of the subcomputations. (This is similar to normal sequential analysis.)

If in parallel, the span is the maximum of the spans of the subcomputations. (This is where analysis of multithreaded algorithms differs.)



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



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

Returning to our example, the span of the parallel recursive calls of P-Fib(n) is:

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

$= \quad T_\infty(n-1) + \Theta(1).$

which has solution $\Theta(n)$.

The parallelism of P-Fib(n) in general (not the specific case we computed earlier) is $T_1(n) / T_\infty = \Theta(F_n/n)$, which grows dramatically, as F_n grows much faster than n.

There is considerable parallel slackness, so above small n there is potential for near perfect linear speedup: there is likely to be something for additional processors to do.

Parallel Loops

So far we have used spawn, but not the **parallel** keyword, which is used with loop constructs such as **for**. Here is an example.

Suppose we want to multiply an n x n matrix $A = (a_{ij})$ by an n-vector $x = (x_j)$. This yields an n-vector $y = (y_i)$ where:

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

for $i = 1, 2, \ldots, n$.

The following algoirthm does this in parallel:

```
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
```

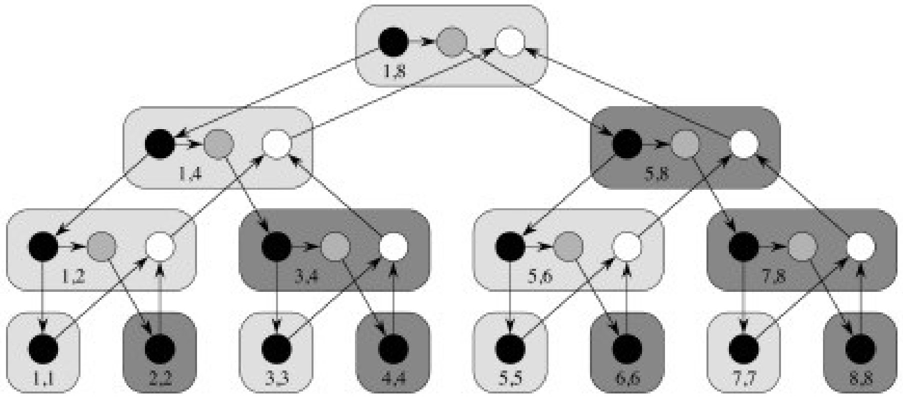
The **parallel for** keywords indicate that each iteration of the loop can be executed concurrently. (Notice that the inner **for** loop is not parallel; a possible point of improvement to be discussed.)

Implementing Parallel Loops

It is not realistic to think that all n subcomputations in these loops can be spawned immediately with no extra work. (For some operations on some hardware up to a constant n this may be possible; e.g., hardware designed for matrix operations; but we are concerned with the general case.) How might this parallel spawning be done, and how does this affect the analysis?

This can be accomplished by a compiler with a divide and conquer approach, itself implemented with parallelism. The procedure shown below is called with Mat-Vec-Main-Loop(A, x, y, n, 1, n). Lines 2 and 3 are the lines originally within the loop.

```
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
```



The computation dag is also shown. It appears that a lot of work is being done to spawn the n leaf node computations, but the increase is not asymptotic.

The *work* of Mat-Vec is $T_1(n) = \Theta(n^2)$ due to the nested loops in 5-7.

Since the tree is a full binary tree, the number of internal nodes is 1 fewer than the leaf nodes, so this extra work is also $\Theta(n)$.

So, the work of recursive spawning contributes a constant factor when amortized across the work of the iterations.

However, concurrency platforms sometimes coarsen the recursion tree by executing several iterations in each leaf, reducing the amount of recursive spawning.

The span is increased by $\Theta(\lg n)$ due to the tree. In some cases (such as this one), this increase is washed out by other dominating factors (e.g., the doubly nested loops).

Nested Parallelism

Returning to our example, the span is $\Theta(n)$ because even with full utilization of parallelism the inner **for** loop still requires $\Theta(n)$. Since the work is $\Theta(n^2)$ the parallelism is $\Theta(n)$. Can we improve on this?

Perhaps we could make the inner **for** loop parallel as well? Compare the original to this revised version:

```
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' (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      parallel for j = 1 to n
7          y_i = y_i + a_ij x_j
8  return y
```

Would it work? We need to introduce a new issue ...

Race Conditions

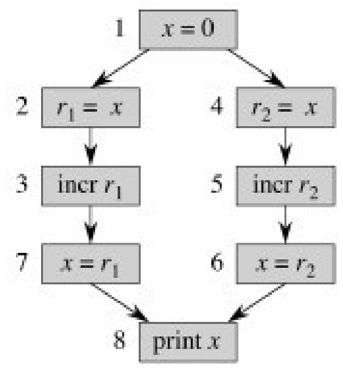
Deterministic algorithms do the same thing on the same input; while **nondeterministic** algorithms may give different results on different runs.

The above Mat-Vec' algorithm is subject to a potential problem called a **determinancy race**: when the outcome of a computation could be nondeterministic (unpredictable). This can happen when two logically parallel computations access the same memory and one performs a write.

Determinancy races are hard to detect with empirical testing: many execution sequences would give correct results. This kind of software bug is consequential: Race condition bugs caused the Therac-25 radiation machine to overdose patients, killing three; and caused the North American Blackout of 2003.

For example, the code shown below might output 1 or 2 depending on the order in which access to x is interleaved by the two threads:

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



step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

After we understand that simple example, let's look at our (renamed) matrix-vector example again:

```
MAT-VEC-WRONG($A, x$)
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      parallel for $j = 1$ to $n$
7          $y_i = y_i + a_{ij} x_j$
8  return $y$
```

Do you see how y_i might be updated differently depending on the order in which parallel invocations of line 7 (including access to current value of y_i and writing new ones) are executed?

Example: Matrix Multiplication

Multithreading the basic algorithm

Here is an algorithm for multithreaded matrix multiplication, based on the $T_1(n) = \Theta(n^3)$ algorithm:

```
P-SQUARE-MATRIX-MULTIPLY($A, B$)
$n = A.rows$
let $C$ be a new $n \times n$ matrix
parallel for $i = 1$ to $n$
    parallel for $j = 1$ to $n$
        $c_{ij} = 0$
        for $k = 1$ to $n$
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return $C$
```

How does this procedure compare to MAT-VEC-WRONG? Is is also subject to a race condition? Why or why not?

The span of this algorithm is $T_\infty(n) = \Theta(n)$, due to the path for spawning the outer and inner parallel loop executions and then the n executions of the innermost **for** loop. So the parallelism is $T_1(n) / T_\infty(n) = \Theta(n^3) / \Theta(n) = \Theta(n^2)$

*Could we get the span down to Θ(1) if we parallelized the inner **for** with **parallel for**?*

Multithreading the divide and conquer algorithm

Here is a parallel version of the divide and conquer algorithm from Chapter 4:

```
P-MATRIX-MULTIPLY-RECURSIVE($C, A, B$)
$n = A.rows$
if $n == 1$
    $c_{11} = a_{11}b_{11}$
else let $T$ be a new $n \times n$ matrix
    partition $A$, $B$, $C$, and $T$ into $n/2 \times n/2$ submatrices
        $A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$; $C_{11}, C_{12}, C_{21}, C_{22}$;
        and $T_{11}, T_{12}, T_{21}, T_{22}$; respectively
    spawn P-MATRIX-MULTIPLY-RECURSIVE($C_{11}, A_{11}, B_{11}$)
    spawn P-MATRIX-MULTIPLY-RECURSIVE($C_{12}, A_{11}, B_{12}$)
    spawn P-MATRIX-MULTIPLY-RECURSIVE($C_{21}, A_{21}, B_{11}$)
    spawn P-MATRIX-MULTIPLY-RECURSIVE($C_{22}, A_{21}, B_{12}$)
    spawn P-MATRIX-MULTIPLY-RECURSIVE($T_{11}, A_{12}, B_{21}$)
    spawn P-MATRIX-MULTIPLY-RECURSIVE($T_{12}, A_{12}, B_{22}$)
    spawn P-MATRIX-MULTIPLY-RECURSIVE($T_{21}, A_{22}, B_{21}$)
    P-MATRIX-MULTIPLY-RECURSIVE($T_{22}, A_{22}, B_{22}$)
    sync
    parallel for $i = 1$ to $n$
        parallel for $j = 1$ to $n$
            $c_{ij} = c_{ij} + t_{ij}$
```

See the text for analysis, which concludes that the work is $\Theta(n^3)$, while the span is $\Theta(\lg^2 n)$. Thus, while the work is the same as the basic algorithm the parallelism is $\Theta(n^3) / \Theta(\lg^2 n)$, which makes good use of parallel resources.

Example: Merge Sort

Divide and conquer algorithms are good candidates for parallelism, because they break the problem into independent subproblems that can be solved separately. We look briefly at merge sort.

Parallelizing Merge-Sort

The dividing is in the main procedure `MERGE-SORT`, and we can parallelize it by spawning the first recursive call:

```
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)
```

`MERGE` remains a serial algorithm, so its work and span are Θ(*n*) as before.

The recurrence for the *work* $MS'_1(n)$ of `MERGE-SORT'` is the same as the serial version:

$$\begin{aligned} MS'_1(n) &= 2MS'_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

The recurrence for the *span* $MS'_\infty(n)$ of `MERGE-SORT'` is based on the fact that the recursive calls run in parallel, so there is only one $n/2$ term:

$$\begin{aligned} MS'_\infty(n) &= MS'_\infty(n/2) + \Theta(n) \\ &= \Theta(n) . \end{aligned}$$

The *parallelism* is thus $MS'_1(n) / MS'_\infty(n) = \Theta(n \lg n / n) = \Theta(\lg n)$.

This is low parallelism, meaning that even for large input we would not benefit from having hundreds of processors. How about speeding up the serial `MERGE`?

Parallelizing Merge

`MERGE` takes two sorted lists and steps through them together to construct a single sorted list. This seems intrinsically serial, but there is a clever way to make it parallel.

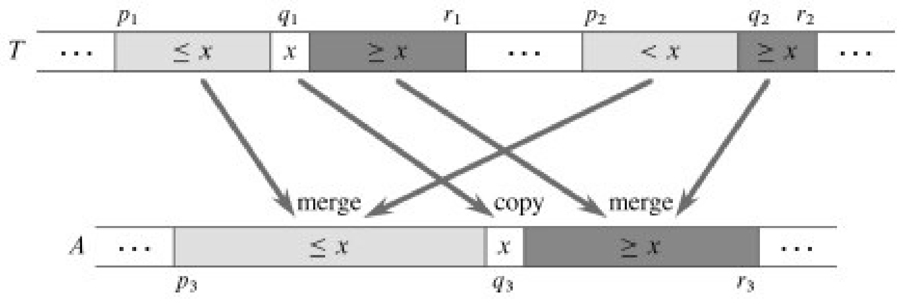
A divide-and-conquer strategy can rely on the fact that they are sorted to break the lists into four lists, two of which will be merged to form the head of the final list and the other two merged to form the tail.

To find the four lists for which this works, we

1. Choose the longer list to be the first list, $T[p_1 .. r_1]$ in the figure below.
2. Find the middle element (median) of the first list (x at q_1).
3. Use binary search to find the position (q_2) of this element if it were to be inserted in the second list

T[p_2 .. r_2].

4. Recursively merge
   - The first list up to just before the median T[p_1 .. q_1-1] and the second list up to the insertion point T[p_2 .. q_2-1].
   - The first list from just after the median T[q_1+1 .. r_1] and the second list after the insertion point T[q_2 .. r_2].
5. Assemble the results with the median element placed between them, as shown below.



The text presents the BINARY-SEARCH pseudocode and analysis of $\Theta(\lg n)$ worst case; this should be review for you. It then assembles these ideas into a parallel merge procedure that merges into a second array Z at location p_3 (r_3 is not provided as it can be computed from the other parameters):

```
P-MERGE($T, p_1, r_1, p_2, r_2, A, p_3$)
 1  $n_1 = r_1 - p_1 + 1$
 2  $n_2 = r_2 - p_2 + 1$
 3  if $n_1 < n_2$                // ensure that $n_1 \ge n_2$
 4      exchange $p_1$ with $p_2$
 5      exchange $r_1$ with $r_2$
 6      exchange $n_1$ with $n_2$
 7  if $n_1 == 0$                 // both empty?
 8      return
 9  else $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
10      $q_2$ = BINARY-SEARCH($T[q_1], T, p_2, r_2$)
11      $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
12      $A[q_3] = T[q_1]$
13      spawn P-MERGE($T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3$)
14      P-MERGE($T, q_1 + 1, r_1, q_2, r_2, A, q_3 + 1$)
15      sync
```

Analysis

My main purpose in showing this to you is to see that even apparently serial algorithms sometimes have a parallel alternative, so we won't get into details, but here is an outline of the analysis:

The span of P-MERGE is the maximum span of a parallel recursive call. Notice that although we divide the first list in half, it could turn out that x's insertion point q_2 is at the beginning or end of the second list. Thus (informally), the maximum recursive span is $3n/4$ (as at best we have "chopped off" 1/4 of the first list).

The text derives the recurrence shown below; it does not meet the Master Theorem, so an approach from a prior exercise is used to solve it:

$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\lg n)$$
$$PM_\infty(n) = \Theta(\lg^2 n). \quad \text{Exercise 4.6-2}$$

Given $1/4 \le \alpha \le 3/4$ for the unknown dividing of the second array, the work recurrence turns out to be:

$$PM_1(n) = PM_1(\alpha n) + PM_1((1-\alpha)n) + O(\lg n)$$

With some more work, $PM_1(n) = \Theta(n)$ is derived. Thus the parallelism is $\Theta(n / \lg^2 n)$

Some adjustment to the `MERGE-SORT'` code is needed to use this `P-MERGE`; see the text. Further analysis shows that the work for the new sort, `P-MERGE-SORT`, is $PMS_1(n \lg n) = \Theta(n)$, and the span $PMS_\infty(n) = \Theta(\lg^3 n)$. This gives parallelism of $\Theta(n / \lg^2 n)$, which is much better than $\Theta(\lg n)$ in terms of the potential use of additional processors as n grows.

The chapter ends with a comment on coarsening the parallelism by using an ordinary serial sort once the lists get small. One might consider whether `P-MERGE-SORT` is still a stable sort, and choose the serial sort to retain this property if it is desirable.

Dan Suthers
Last modified: Sun Apr 12 02:17:14 HST 2015
Images are from the instructor's material for Cormen et al. Introduction to Algorithms, Third Edition.