Managing Multiple Mutable Data Structures

COS 326
David Walker
Princeton University
We explored two programming disciplines that help us manage parallelism and concurrency:

- **Futures:**
  - future : ('a -> 'b) -> 'a -> 'b future
  - force : 'a future -> 'a
  - create a future to run a function in the background
  - useful in divide-and-conquer parallel programming

- **Mutexes:**
  - with_lock : mutex -> (unit -> 'b) -> 'b
  - associate each mutable data structure with a lock m
  - protect all accesses to a mutable data structure with with_lock m

This Time: Sometimes a computation depends upon several mutable data structures.

- eg: to transfer a balance from one bank account to another
- our existing techniques break down
Another Example

type 'a stack = { mutable contents : 'a list;
    lock : Mutex.t
 };;

let empty () = {contents=[]; lock=Mutex.create()};;

let push (s:'a stack) (x:'a) : unit =
    with_lock s.lock (fun _ ->
        s.contents <- x::s.contents)
;;

let pop (s:'a stack) : 'a option =
    with_lock s.lock (fun _ ->
        match s.contents with
            | [] -> None
            | h::t -> (s.contents <- t ; Some h))
;;
Another Example

```ocaml
type 'a stack = { mutable contents : 'a list;
                  lock : Mutex.t }

val empty : () -> 'a stack
val push : 'a stack -> a -> unit
val pop : 'a stack -> 'a option

let transfer_one (s1:'a stack) (s2: 'a stack) =
  with_lock s1.lock (fun _ ->
      match pop s1 with
      | None -> ()
      | None -> ()
      | Some x -> push s2 x)
```
Another Example

type 'a stack = { mutable contents : 'a list;
    lock : Mutex.t }

val empty : () -> 'a stack
val push : 'a stack -> a -> unit
val pop : 'a stack -> 'a option

let transfer_one (s1:'a stack) (s2: 'a stack) =
    with_lock s1.lock (fun _ ->
        match pop s1 with
        | None -> ()
        | Some x -> push s2 x)

Unfortunately, we already hold s1.lock when we invoke pop s1 which tries to acquire the lock.
Another Example

type 'a stack = { mutable contents : 'a list;
    lock : Mutex.t }

val empty : () -> 'a stack
val push : 'a stack -> a -> unit
val pop : 'a stack -> 'a option

let transfer_one (s1:'a stack) (s2: 'a stack) =
    with_lock s1.lock (fun _ ->
        match pop s1 with
        | None -> ()
        | Some x -> push s2 x)

Unfortunately, we already hold s1.lock when we invoke pop s1 which tries to acquire the lock.

So we end up dead-locked.
Another Example

type 'a stack = { mutable contents : 'a list;
    lock : Mutex.t }

val empty : () -> 'a stack
val push : 'a stack -> a -> unit
val pop : 'a stack -> 'a option

let transfer_one (s1: 'a stack) (s2: 'a stack) =
    with_lock s1.lock (fun _ ->
        match pop s1 with
        | None -> ()
        | Some x -> push s2 x)

Avoid deadlock by deleting the line that acquires s1.lock initially
A trickier problem

```ocaml
type 'a stack = { mutable contents : 'a list;
                 lock : Mutex.t }

val empty : () -> 'a stack
val push : 'a stack -> a -> unit
val pop  : 'a stack -> 'a option

let pop_two (s1:'a stack)
             (s2:'a stack) : ('a * 'a) option =
  match pop s1, pop s2 with
  | Some x, Some y -> Some (x,y)
  | Some x, None   -> push s1 x ; None
  | None, Some y   -> push s2 y  ; None
```

Either:

(1) pop one from each if both nonempty, or

(2) have no effect at all
A trickier problem

type 'a stack = { mutable contents : 'a list;
                lock : Mutex.t }

val empty : () -> 'a stack
val push : 'a stack -> a -> unit
val pop  : 'a stack -> 'a option

let pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =
  match pop s1, pop s2 with
  | Some x, Some y -> Some (x,y)
  | Some x, None -> push s1 x ; None
  | None, Some y -> push s2 y ; None

But some other thread could sneak in here and try to perform an operation on our contents before we’ve managed to push the value back on.
let no_lock_pop (s1:'a stack) : 'a option =  
match s1.contents with  
| [] -> None  
| h::t -> (s1.contents <- t ; Some h)

let no_lock_push (s1:'a stack) (x :'a) : unit =  
contents <- x::contents

let pop_two (s1:'a stack)  
(s2:'a stack) : ('a * 'a) option =  
with_lock s1.lock (fun _ ->  
with_lock s2.lock (fun _ ->  
match no_lock_pop s1, no_lock_pop s2 with  
| Some x, Some y -> Some (x,y)  
| Some x, None -> no_lock_push s1 x ; None  
| None, Some y -> no_lock_push s2 y ; None))
Yet another broken solution

```ocaml
let no_lock_pop (s1:'a stack) : 'a option =
  match s1.contents with
  | [] -> None
  | h::t -> (s1.contents <- t ; Some h)

let no_lock_push (s1:'a stack) (x :'a) : unit =
  contents <- x:::contents

let pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =
  with_lock s1.lock (fun _ ->
    with_lock s2.lock (fun _ ->
      match no_lock_pop s1, no_lock_pop s2 with
      | Some x, Some y -> Some (x,y)
      | Some x, None -> no_lock_push s1 x ; None
      | None, Some y -> no_lock_push s2 y ; None
    ))
```
Yet another broken solution

```ocaml
let no_lock_pop (s1:'a stack) : 'a option =  
  match s1.contents with  
  | [] -> None  
  | h::t -> (s1.contents <- t ; Some h)

let no_lock_push (s1:'a stack) (x :'a) : unit =  
  s1.contents <- x::s1.contents

let pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =  
  with_lock s1.lock (fun _ ->  
    with_lock s2.lock (fun _ ->  
      match no_lock_pop s1, no_lock_pop s2 with  
      | Some x, Some y -> Some (x,y)  
      | Some x, None -> no_lock_push s1 x ; None  
      | None, Some y -> no_lock_push s2 y ; None))
```

What happens if we call `pop_two x x`?
Yet another broken solution

```ocaml
let no_lock_pop (s1:'a stack) : 'a option =
  match s1.contents with
  | [] -> None
  | h::t -> (s1.contents <- t ; Some h)

let no_lock_push (s1:'a stack) (x : 'a) : unit =
  contents <- x::contents

let pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =
  with_lock s1.lock (fun _ ->
    with_lock s2.lock (fun _ ->
      match no_lock_pop s1, no_lock_pop s2 with
      | Some x, Some y -> Some (x,y)
      | Some x, None -> no_lock_push s1 x ; None
      | None, Some y -> no_lock_push s2 y ; None)))
```

What happens if two threads are trying to call `pop_two` at the same time?

In particular, consider:

```ocaml
Thread.create (fun _ -> pop_two x y)
Thread.create (fun _ -> pop_two y x)
```
Yet another broken solution

```ocaml
let no_lock_pop (s1:'a stack) : 'a option = 
match s1.contents with 
| [] -> None 
| h::t -> (s1.contents <- t ; Some h)

let no_lock_push (s1:'a stack) (x :'a) : unit = 
contents <- x::contents

let pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option = 
with_lock s1.lock (fun _ -> 
with_lock s2.lock (fun _ -> 
match no_lock_pop s1, no_lock_pop s2 with 
| Some x, Some y -> Some (x,y) 
| Some x, None -> no_lock_push s1 x ; None 
| None, Some y -> no_lock_push s2 y ; None))
```

More general problem:

- `Thread.create (fun _ -> pop_two x y)`
- `Thread.create (fun _ -> pop_two y x)`

One possible interleaving:
- T1 acquires x’s lock.
- T2 acquires y’s lock.
- T1 tries to acquire y’s lock and blocks.
- T2 tries to acquire x’s lock and blocks.

**DEADLOCK**
type 'a stack = { mutable contents : 'a list; lock : Mutex.t; id : int }  

let new_id : unit -> int =  
  let c = ref 0 in (fun _ -> c := (!c) + 1 ; !c)  

let empty () = {contents=[]; lock=Mutex.create(); id=new_id()};;  

let no_lock_pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =  
  match no_lock_pop s1, no_lock_pop s2 with  
  | Some x, Some y -> Some (x,y)  
  | Some x, None -> no_lock_push s1 x; None  
  | None, Some y -> no_lock_push s2 y; None  

let pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =  
  if s1.id < s2.id then  
    with_lock s1.lock (fun _ ->  
      with_lock s2.lock (fun _ ->  
        no_lock_pop_two s1 s2))  
  else if s1.id > s2.id then  
    with_lock s2.lock (fun _ ->  
      with_lock s1.lock (fun _ ->  
        no_lock_pop_two s1 s2))  
  else with_lock s1.lock (fun _ -> no_lock_pop_two s1 s2)
type 'a stack = { mutable contents : 'a list; lock : Mutex.t; id : int }

let new_id : unit -> int =
  let c = ref 0 in let l = Mutex.create() in
  (fun _ -> with_lock l (fun _ -> (c := (!c) + 1 ; !c)))

let empty () = {contents=[]; lock=Mutex.create(); id=new_id()};;

let no_lock_pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =
  match no_lock_pop s1, no_lock_pop s2 with
  | Some x, Some y -> Some (x,y)
  | Some x, None -> no_lock_push s1 x; None
  | None, Some y -> no_lock_push s2 y; None

let pop_two (s1:'a stack) (s2:'a stack) : ('a * 'a) option =
  ...
  ;;
Refined Design Pattern

• **Associate a lock with each shared, mutable object.**

• **Choose some ordering on shared mutable objects.**
  – doesn’t matter what the order is, as long as it is total.
  – in C/C++, often use the address of the object as a unique number.
  – Our solution: *add a unique ID number to each object*

• **To perform actions on a set of objects S atomically:**
  – acquire the locks for the objects in S *in order.*
  – perform the actions.
  – release the locks.
Refined Design Pattern

• Associate a lock with each shared, mutable object.
• Choose some ordering on shared mutable objects.
  – doesn’t matter what the order is, as long as it is total.
  – in C/C++, often use the address of the object as a unique number.
  – Our solution: add a unique ID number to each object.
• To perform actions on a set of objects in a program:
  – acquire the locks for the objects in \( S \) in order.
  – perform the actions.
  – release the locks.

Important!
Acquire all the locks you will need \textbf{BEFORE} performing any irreversible actions!

BUT: IN A BIG PROGRAM, IT IS REALLY HARD TO GET THIS RIGHT
A HUGE COMPONENT OF PL RESEARCH INVOLVES TRYING TO FIND THE MISTAKES PEOPLE MAKE WHEN DOING THIS. AVOID WHenever Possible! USE FUNCTIONAL ABSTRACTIONS!
SUMMARY
Reasoning about the correctness of pure parallel programs that include futures is easy -- no harder than ordinary, sequential programs. (Reasoning about their performance may be harder.)

Reasoning about shared variables and semaphores is hard in general, but futures are a discipline for getting it right.

Much of programming-language design is the art of finding good disciplines in which it’s harder* to write bad programs.

Even aside from PL design, the same is true of software engineering with Abstract Data Types: engineer disciplines in your interfaces, harder for the user to get it wrong.

*but somebody will always find a way...
Programming with mutation, threads and locks

Reasoning about the correctness of pure parallel programs that include futures is easy -- no harder than ordinary, sequential programs. (Reasoning about their performance may be harder.)

Reasoning about concurrent programs with *effects* requires considering *all interleavings* of instructions of concurrently executing threads.

- often too many interleavings for normal humans to keep track of
- nonmodular: you often have to look at the details of each thread to figure out what is going on
- locks cut down interleavings
- but knowing you have done it right still requires deep analysis

*and worse...*
Scheduling Parallel Computations
let x = 1 + 2 in
3 + x
let \( x = 1 + 2 \) in

\[
3 + x
\]

\[
\begin{align*}
x &= 1 + 2 \\
3 + x &\quad \text{cost} = 1
\end{align*}
\]
let \( x = 1 + 2 \) in
\( 3 + x \)

dependence:
\( x = 1 + 2 \) happens before \( 3 + x \)
Execution of dependency diagrams: A processor can only begin executing the computation associated with a block when the computations of all of its predecessor blocks have been completed.
step 1:
execute first block

\[ x = 1 + 2 \]
\[ 3 + x \]

cost = 1

cost = 1

Cost so far: 0
step 1: execute first block

\[
\begin{align*}
    x &= 1 + 2 \\
    3 + x &= \text{cost} = 1
\end{align*}
\]

Cost so far: 1
step 2: execute second block because all of its predecessors have been completed

Cost so far: 1
Visualizing Computational Costs

step 2: execute second block because all of its predecessors have been completed

Cost so far: 1 + 1
let \( x = 1 + 2 \) in

\[
\begin{align*}
3 + x &= 1 + 2 \\
\text{cost} &= 1
\end{align*}
\]

\[
\begin{align*}
3 + x &= 1 + 2 \\
\text{cost} &= 1
\end{align*}
\]

\[
\begin{align*}
\text{total cost} &= 1 + 1 \\
&= 2
\end{align*}
\]
Visualizing Computational Costs

(1 + 2 || f 3)

parallel pair:
compute both left and right-hand sides independently
return pair of values
(easy to implement using futures)
Visualizing Computational Costs

\[(1 + 2 || f 3)\]

The diagram represents a computational cost analysis with nodes A, B, C, and D, where:

- Node A: cost = 1
- Node B: cost = 1, operation: 1 + 2
- Node C: cost = 7, operation: f 3
- Node D: cost = 1, operation: (,)

The edges connect these nodes, indicating the computational flow and costs.
Suppose we have 1 processor. How much time does this computation take?
Suppose we have 1 processor. How much time does this computation take? Scheduld A-B-C-D: 1 + 1 + 7 + 1
Suppose we have 1 processor. How much time does this computation take?
Schedule A-C-B-D: $1 + 1 + 7 + 1$
Suppose we have 2 processors. How much time does this computation take?
Visualizing Computational Costs

Suppose we have 2 processors. How much time does this computation take? Cost so far: 1
Suppose we have 2 processors. How much time does this computation take? Cost so far: $1 + \max(1,7)$
Suppose we have 2 processors. How much time does this computation take? Cost so far: $1 + \max(1,7) + 1$
Visualizing Computational Costs

\[(1 + 2 || f 3)\]

\[\begin{align*}
\text{cost} &= 1 \\
A &
\end{align*}\]

\[\begin{align*}
\text{cost} &= 1 \\
B &
\end{align*}\]

\[\begin{align*}
\text{cost} &= 7 \\
C &
\end{align*}\]

\[\begin{align*}
\text{cost} &= 1 \\
D &
\end{align*}\]

Suppose we have 2 processors. How much time does this computation take? Total cost: \(1 + \max(1,7) + 1\). We say the schedule we used was: A-CB-D
Suppose we have 3 processors. How much time does this computation take?
Suppose we have 3 processors. How much time does this computation take? Schedule A-BC-D: 1 + max(1,7) + 1 = 9
Suppose we have infinite processors. How much time does this computation take? Schedule A-BC-D: \( 1 + \max(1,7) + 1 = 9 \)
Work and Span

• Understanding the complexity of a parallel program is a little more complex than a sequential program
  – the number of processors has a significant effect

• One way to *approximate* the cost is to consider a parallel algorithm independently of the machine it runs on is to consider *two* metrics:
  – **Work**: The cost of executing a program with just 1 processor.
  – **Span**: The cost of executing a program with an infinite number of processors

• Always good to minimize work
  – Every instruction executed consumes energy
  – Minimize span as a second consideration
  – Communication costs are also crucial (we are ignoring them)
Parallelism

The **parallelism** of an algorithm is an estimate of the maximum number of processors an algorithm can profit from.

- parallelism = work / span

If work = span then parallelism = 1.
- We can only use 1 processor
- It's a sequential algorithm

If span = ½ work then parallelism = 2
- We can use up to 2 processors

If work = 100, span = 1
- All operations are independent & can be executed in parallel
- We can use up to 100 processors
Series-parallel graphs arise from execution of functional programs with parallel pairs. Also known as well-structured, nested parallelism.
In general, a series-parallel graph has a source and a sink and is:
- a single node, or
- two series-parallel graphs in sequence, or
- two series-parallel graphs in parallel
Not a Series-Parallel Graph

However:
The results about greedy schedulers (next few slides) do apply to DAG schedules as well as series-parallel schedules!
Let's assume each node costs 1.

**Work**: sum the nodes.

**Span**: longest path from source to sink.
Work and Span of Acyclic Graphs

Let's assume each node costs 1.

**Work**: sum the nodes.

**Span**: longest path from source to sink.

work = 10
span = 5
Scheduling

Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
E H
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
E H
I
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
E H
I
J
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
E H
I
J
F
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
E H H I
F
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
E H  H I
J  E J
F
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
E H
H I
J
E J
F
F
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
E H
H I
J
F

Conclusion:
How you schedule jobs can have an impact on performance.
Greedy Schedulers

• Greedy schedulers will schedule some task to a processor as soon as that processor is free.
  – Doesn't sound so smart!

• Properties (for p processors):
  – $T(p) < \frac{\text{work}}{p} + \text{span}$
    • won't be worse than dividing up the data perfectly between processors, except for the last little bit, which causes you to add the span on top of the perfect division

  – $T(p) \geq \max(\frac{\text{work}}{p}, \text{span})$
    • can't do better than perfect division between processors (work/p)
    • can't be faster than span
Greedy Schedulers

Properties (for \( p \) processors):

\[
\max(\text{work}/p, \text{span}) \leq T(p) < \text{work}/p + \text{span}
\]

Consequences:

– as \( \text{span} \) gets small relative to \( \text{work}/p \)
  
  • \( \text{work}/p + \text{span} \Rightarrow \text{work}/p \)
  
  • \( \max(\text{work}/p, \text{span}) \Rightarrow \text{work}/p \)
  
  • so \( T(p) \Rightarrow \text{work}/p \) -- greedy schedulers converge to the optimum!

– if \( \text{span} \) approaches the work
  
  • \( \text{work}/p + \text{span} \Rightarrow \text{span} \)
  
  • \( \max(\text{work}/p, \text{span}) \Rightarrow \text{span} \)
  
  • so \( T(p) \Rightarrow \text{span} \) – greedy schedulers converge to the optimum!
COMPLEXITY OF PARALLEL PROGRAMS
Divide-and-Conquer Parallel Algorithms

• Split your input in 2 or more subproblems
• Solve the subproblems recursively in parallel
• Combine the results to solve the overall problem
let rec mergesort (l : int list) : int list =
  match l with
  | []       -> []
  | [x]      -> [x]
  | _        ->
    let (pile1,pile2) = split l in
    let (sorted1,sorted2) =
      both mergesort pile1
      mergesort pile2
    in
    merge sorted1 sorted2
  ;;

for sequential mergesort, replace with: (mergesort sorted1, mergesort sorted2)
let rec split l =
  match l with
  | []  -> ([], [])
  | [x] -> ([x], [])
  | x :: y :: xs ->
    let (pile1, pile2) = split xs in
    (x :: pile1, y :: pile2)

let rec merge l1 l2 =
  match (l1, l2) with
  | ([], l2) -> l2
  | (l1, []) -> l1
  | (x :: xs, y :: ys) ->
    if x < y then
      x :: merge xs l2
    else
      y :: merge l1 ys
Complexity

let rec mergesort (l : int list) : int list =
    match l with
    []  -> []
| [x] -> [x]
| _   ->
    let (pile1,pile2) = split l in
    let (sorted1,sorted2) =
        both mergesort pile1
        mergesort pile2
    in
    merge sorted1 sorted2

Assume input list of size n:
work_mergesort(n) = work_split(n)
                    + 2*work_mergesort(n/2)
                    + work_merge(n)
let rec mergesort (l : int list) : int list =
    match l with
    | []  -> []
    | [x] -> [x]
    | _   ->
        let (pile1,pile2) = split l in
        let (sorted1,sorted2) =
            both mergesort pile1
            mergesort pile2
        in
        merge sorted1 sorted2

Assume input list of size n:
work_mergesort(n) = work_split(n)
    + 2*work_mergesort(n/2)
    + work_merge(n)

= k1*n
+ 2*work_mergesort(n/2)
+ k2*n

read this as
"approximately equal to"
Complexity

let rec mergesort (l : int list) : int list =
  match l with
  | []    -> []
  | [x]   -> [x]
  | _     ->
    let (pile1,pile2) = split l in
    let (sorted1,sorted2) =
        both mergesort pile1
        mergesort pile2
    in
    merge sorted1 sorted2

Assume input list of size $n$:

$$\text{work\_mergesort}(n) = \text{work\_split}(n) = k\times n$$
$$+ 2\times \text{work\_mergesort}(n/2)$$
$$+ \text{work\_merge}(n)$$
let rec mergesort (l : int list) : int list =
  match l with
  | []    -> []
  | [x]   -> [x]
  | _     ->
    let (pile1,pile2) = \texttt{split} l in
    let (sorted1,sorted2) =
      \textbf{both} mergesort pile1
      mergesort pile2
    in
    \texttt{merge} sorted1 sorted2

Assume input list of size $n$:
\begin{align*}
\text{work\_mergesort}(n) &= \text{work\_split}(n) \\
&\quad + 2\times\text{work\_mergesort}(n/2) \\
&\quad + \text{work\_merge}(n) \\
&= k\times n \\
&\quad + 2\times\text{work\_mergesort}(n/2) \\
&= O(n \log n)
\end{align*}
let rec mergesort (l : int list) : int list =
  match l with
  []  -> []
| [x] -> [x]
| _   ->
    let (pile1,pile2) = split l in
    let (sorted1,sorted2) =
      both mergesort pile1
      mergesort pile2
    in
    merge sorted1 sorted2

Assume input list of size n:
span_mergesort(n) = span_split(n)
  + \text{max}(\text{span}_\text{mergesort}(n/2), \text{span}_\text{mergesort}(n/2))
  + \text{span}_\text{merge}(n)
let rec mergesort (l : int list) : int list = 
match l with 
  [] -> []
| [x] -> [x]
| _  -> 
  let (pile1,pile2) = split l in 
  let (sorted1,sorted2) = 
    both mergesort pile1 
    mergesort pile2 
  in 
  merge sorted1 sorted2

Assume input list of size n:
span_mergesort(n) = k*n
+ span_mergesort(n/2)
let rec mergesort : int list -> int list = 
    match l with 
    | []    -> [] 
    | [x]   -> [x] 
    | _     -> 
        let (pile1, pile2) = split l in 
        let (sorted1, sorted2) = 
            both mergesort pile1 
            mergesort pile2 
        in 
        merge sorted1 sorted2 

Assume input list of size n: 
\[ \text{span}_\text{mergesort}(n) = k \times n \]
\[ + k \times (n/2 + n/4 + n/8 + ...) \]
let rec mergesort (l : int list) : int list =
    match l with
    []  -> []
  | [x] -> [x]
  | _   ->
    let (pile1,pile2) = \textbf{split} l in
    let (sorted1,sorted2) =
      \textbf{both} mergesort pile1
      mergesort pile2
    in
    \textbf{merge} sorted1 sorted2

Assume input list of size n:
\text{span\_mergesort}(n) = 2 \times k \times n
= O(n)
let rec mergesort (l : int list) : int list =
  match l with
  | []  -> []
  | [x] -> [x]
  | _   ->
      let (pile1,pile2) = \textit{split} l in
      let (sorted1,sorted2) =
          \textit{both} mergesort pile1
          mergesort pile2
      in
      \textit{merge} sorted1 sorted2

\textbf{Summary for input list of size }n:\textbf{ parallelism?}
\begin{align*}
\text{work\_mergesort}(n) &= k*n*log n \\
\text{span\_mergesort}(n) &= k*n
\end{align*}
let rec mergesortsort (l : int list) : int list =
match l with
  []  -> []
| [x] -> [x]
| _   ->
  let (pile1,pile2) = split l in
  let (sorted1,sorted2) = both mergesort pile1
                               mergesort pile2
  in
  merge sorted1 sorted2

Summary for input list of size n:
work_mergesort(n) = k*n*log n
span_mergesort(n) = k*n

parallelism?
parallelism = work/span
            = n*log n / n
            = log n

when sorting 10 billion entries, can only make use of 30 machines
let rec mergesort (l : int list) : int list =
  match l with
  []     -> []
  | [x]   -> [x]
  | _      ->
    let (pile1,pile2) = split l in
    let (sorted1,sorted2) =
      both mergesort pile1
      mergesort pile2
    in
    merge sorted1 sorted2

Summary for input list of size n:
work_mergesort(n) = k*n*log n
span_mergesort(n) = k*n

splitting and merging take linear time – too long to get good speedups

parallelism?
parallelism = work/span
            = n*log n / n
            = log n

when sorting 10 billion entries, can only make use of 30 machines
Complexity

when sorting 10 billion entries, can only make use of 30 machines/cores
data centers have 10s of 1000s of machines or more

Problem: splitting and merging take linear time – too long to get good speedups

Problem: cutting a list in half takes at least time proportional to n/2

Problem: stitching 2 lists together of size n/2 takes n/2 time

Conclusion: lists are a bad data structure to choose
Consider balanced trees:

- Splitting is pretty easy in constant time.
- Merging is harder, but can be done in poly-log time.
type tree = Empty | Node of tree * int * tree

let node left i right = Node (left, i, right)

let one i = node Empty i Empty

• Problem: Given a balanced tree t, return a balanced tree with the same elements, in order:
  – elements in the left subtree are less than the root
  – elements in the right subtree are greater than the root
type tree = Empty | Node of tree * int * tree

let node left i right = Node (left, i, right)

let one i = node Empty i Empty

let rec tsort t =
  match t with
  Empty -> Empty |
  Node (l, i, r) ->
  let (l', r') = both tsort l
  tsort r
  in
  rebalance (merge (merge l' r') (one i))
Parallel TreeSort

```
type tree = Empty | Node of tree * int * tree

let node left i right = Node (left, i, right)

let one i = node Empty i Empty

let rec tsort t =
  match t with
  | Empty -> Empty
  | Node (l, i, r) ->
    let (l', r') = both tsort l
    tsort r
  in
  merge (merge l' r') (one i)
```
Merging trees

• Subproblem: Given two sorted, balanced trees, l and r, create a new tree with the same elements that is also balanced and whose elements are in order.
• Uses \texttt{split\_at} \texttt{t i}
  — divides \texttt{t} into items less than \texttt{i} and items greater than \texttt{i}

\begin{verbatim}
let rec merge (t1:tree) (t2:tree) : tree =
  match t1 with
  | Empty -> t2
  | Node (l1, i, r1) ->
    let (l2, r2) = \texttt{split\_at} t2 i in
    let (t1', t2') = both (merge l1) l2
    (merge r1) r2
    in
    Node (t1', i, t2')
\end{verbatim}
• Sub-problem: Divide t in to items less than i and items greater than i

let rec split_at t bound =
    match t with
    | Empty -> (Empty, Empty)
    | Node (l, i, r) ->
        if bound < i then
            let (ll, lr) = split_at l bound in
            (ll, Node (lr, i, r))
        else
            let (rl, rr) = split_at r bound in
            (Node (l, i, rl), rr)
Splitting a tree

- Sub-problem: Divide t in to items less than i and items greater than i

```ocaml
let rec split_at t bound =
  match t with
  | Empty -> (Empty, Empty)
  | Node (l, i, r) ->
    if bound < i then
      let (ll, lr) = split_at l bound in
      (ll, Node (lr, i, r))
    else
      let (rl, rr) = split_at r bound in
      (Node (l, i, rl), rr)

span (h) = k*h
```

where h is the height of the tree t
h = log(n) if t is balanced with n nodes
let rec merge (t1:tree) (t2:tree) : tree =
  match t1 with
    Empty -> t2
  | Node (l1, i, r1) ->
    let (l2, r2) = split_at t2 i in
    let (t1', t2') = both (merge l1) l2 (merge r1) r2 in
    Node (t1', i, t2')

let's assume t1 and t2 are balanced and have heights h1, h2 and h1 >= h2:

  span_merge(h1,h2)
= span_split(h2) + max(span_merge(h1-1), span_merge(h2-1))
= k*h2 + span_merge(h1-1)
= k*h2*h1
let rec tsort t =
    match t with
    Empty -> Empty
  | Node (l, i, r) ->
    let (l', r') = both tsort l
                    tsort r
    in
    merge (merge l' r') (one i)

let's assume:
• t is balanced with n nodes and height h = log n
• tsort returns balanced trees (l', r')
• merge returns balanced trees
let rec tsort t =
match t with
  Empty -> Empty
| Node (l, i, r) ->
  let (l', r') = both tsort l
tsort r
in
merge (merge l' r') (one i)

let's assume:
• t is balanced with n nodes and height h = log n
• tsort returns balanced trees (l', r')
• merge returns balanced trees

span_tsort(h)
= max(span_tsort(h-1),
    span_tsort(h-1))
+ span_merge(h-1,h-1)
+ span_merge(h,1)
let rec tsort t =
  match t with
  | Empty -> Empty
  | Node (l, i, r) ->
    let (l', r') = both tsort l
                 tsort r
    in
    merge (merge l' r') (one i)

let's assume:
  • t is balanced with n nodes and height h = \log n
  • tsort returns balanced trees (l', r')
  • merge returns balanced trees

\[
\text{span}_\text{tsort}(h) = \max(\text{span}_\text{tsort}(h-1), \text{span}_\text{tsort}(h-1)) + \text{span}_\text{merge}(h-1,h-1) + \text{span}_\text{merge}(h,1)
= \text{span}_\text{tsort}(h-1) + k(h-1)(h-1) + k\cdot h
\]
let rec tsort t =
  match t with
  | Empty -> Empty
  | Node (l, i, r) ->
    let (l', r') = both tsort l
                 tsort r
    in
    merge (merge l' r') (one i)

let's assume:
  • t is balanced with n nodes and height h = log n
  • tsort returns balanced trees (l', r')
  • merge returns balanced trees

span_tsort(h)
  = max(span_tsort(h-1),
        span_tsort(h-1))
  + span_merge(h-1,h-1)
  + span_merge(h,1)
  = span_tsort(h-1)
  + k*(h-1)*(h-1) + k*h
  = k*h*h*h
let rec tsort t =
  match t with
  | Empty -> Empty
  | Node (l, i, r) ->
    let (l', r') = both tsort l tsort r
    in
    merge (merge l' r') (one i)

let's assume:
- t is balanced with n nodes and height h = log n
- tsort returns balanced trees (l', r')
- merge returns balanced trees

\[
\text{span}_\text{tsort}(h) = \max(\text{span}_\text{tsort}(h-1), \text{span}_\text{tsort}(h-1)) + \text{span}_\text{merge}(h-1,h-1) + \text{span}_\text{merge}(h,1) = \text{span}_\text{tsort}(h-1) + k*(h-1)*(h-1) + k*h = k*h*h*h = O(\log^3 n)
\]
A Summary of Parallel Sorting Exercise

Both parallel list sort and parallel tree sort follow a traditional parallel divide-and-conquer strategy.

By changing data structures from lists to trees, we were able to:

- split our data in half in **constant span** instead of **linear span**
- merge our data back together in $\log^3 n$ span instead of **linear span**

We get more parallelism:

- with lists: $\text{work/span} = \log n$
  - make use of 30 machines when sorting 10 billion items
- with trees: $\text{work/span} = \frac{n \log n}{\log^3 n} = \frac{n}{\log^2 n}$
  - make use of millions* of machines when sorting 10 billion items

  - **caveat:** we didn't factor in data communication costs!

*Well, almost. What is $\log_2(10,000,000,000)$?*
Series parallel-graphs describe the kinds of control structures that arise in pure functional programs with structured, parallel fork-join execution

- **Work**: total number/cost of operations
  - time program execution takes with 1 processor
  - \[ \text{Work}( e_1 || e_2 ) = \text{Work}(e_1) + \text{Work}(e_2) + 1 \]

- **Span**: length of the longest dependency chain
  - time program execution takes with infinite processors
  - \[ \text{Span} ( e_1 || e_2 ) = \max ( \text{Span} e_1, \text{Span} e_2 ) + 1 \]

- **Parallelism**: Work / Span

Many parallel algorithms follow a divide-and-conquer strategy
- efficient algorithms divide quickly and merge quickly
Parallel Collections
Parallel Collections

One way to give programmers access to parallelism in a functional style (even in an imperative language) is to develop a library for programming parallel collections.

Example collections: sets, tables, dictionaries, sequences
Example bulk operations: create, map, reduce, join, filter
Parallel Sequences

- Parallel sequences

  \(< e_1, e_2, e_3, ..., e_n >\)

- Languages:
  - Nesl [Blelloch]
  - Data-parallel Haskell
Parallel Sequences: Selected Operations

\[
\text{tabulate} : (\text{int} \rightarrow 'a) \rightarrow \text{int} \rightarrow 'a \text{ seq}
\]

\[
\text{tabulate } f \ n = \langle f \ 0, \ f \ 1, \ \ldots, \ f \ (n-1) \rangle
\]

\[
\text{work} = O(n \cdot \text{work}(f)) \quad \text{span} = O(1 \cdot \text{span}(f))
\]
Parallel Sequences: Selected Operations

```plaintext

tabulate : (int -> 'a) -> int -> 'a seq

tabulate f n  == <f 0, f 1, ..., f (n-1)>
work = O(n·work(f))   span = O(1·span(f))

nth : 'a seq -> int -> 'a

nth <e0, e1, ..., e(n-1)> i == ei
work = O(1)   span = O(1)
```
Parallel Sequences: Selected Operations

\[
\text{tabulate : } (\text{int} \rightarrow 'a) \rightarrow \text{int} \rightarrow 'a \text{ seq}
\]

gives:

\[
\text{tabulate } f \ n \ = \ <f \ 0, \ f \ 1, \ ..., \ f \ (n-1)>
\]

work = \(O(n \cdot \text{work(f)})\) \quad \text{span} = \(O(1 \cdot \text{span(f)})\)

\[
\text{nth : } 'a \text{ seq} \rightarrow \text{int} \rightarrow 'a
\]

gives:

\[
\text{nth } <e_0, \ e_1, \ ..., \ e_{(n-1)}> \ i \ = \ e_i
\]

work = \(O(1)\) \quad \text{span} = \(O(1)\)

\[
\text{length : } 'a \text{ seq} \rightarrow \text{int}
\]

gives:

\[
\text{length } <e_0, \ e_1, \ ..., \ e_{(n-1)}> \ = \ n
\]

work = \(O(1)\) \quad \text{span} = \(O(1)\)
Problems

Write a function that creates the sequence <0, ..., n-1>

Write a function such that given a sequence <v0, ..., vn-1>, maps f over each element of the sequence.
Work = O(n); Span = O(1)  (if f is a constant-work function)

Write a function such that given a sequence <v1, ..., vn-1>, reverses the sequence.
Work = O(n); Span = O(1)

Try it!

Operations:

- tabulate f n
- nth i s
- length s
Solutions

(* create n == <0, 1, ..., n-1> *)
let create n =
  tabulate (fun i -> i) n

(* map f <v0, ..., vn-1> == <f v0, ..., f vn-1> *)
let map f s =
  tabulate (fun i -> f (nth s i)) (length s)

(* reverse <v0, ..., vn-1> == <vn-1, ..., v0> *)
let reverse f s =
  let n = length s in
  tabulate (fun i -> nth s (n-i-1)) n
One more problem

- Consider the problem of determining whether a sequence of parentheses is balanced or not. For example:
  - balanced: ()()(())
  - not balanced: ( or ) or ()

- Try formulating a divide-and-conquer parallel algorithm to solve this problem efficiently:

```ocaml
type paren = L | R (* L(eft) or R(ight) paren *)

let balanced (ps : paren list) : bool = ...
```

- You will need another function on sequences:

```ocaml
(* split s n divides s into (s1, s2) such that s1 is the first n elements of s and s2 is the rest
  Work = O(n) Span = O(1) *)
split : 'a sequence -> int -> 'a sequence * 'a sequence
```