Parallelism and Concurrency (Part II)

COS 326
David Walker
Princeton University
let x = 1 + 2 in
3 + x
let x = 1 + 2 in
3 + x

x = 1 + 2

3 + x
cost = 1
cost = 1
let \( x = 1 + 2 \) in \( 3 + x \)

dependence:
\( x = 1 + 2 \) happens before \( 3 + x \)
let $x = 1 + 2$ in $3 + x$

$\begin{align*}
\text{cost} &= 1 \\
3 + x &\quad \text{cost} = 1 \\
x = 1 + 2 &\quad \text{cost} = 1 \\
\text{total cost} &= 1 + 1 \\
&= 2
\end{align*}$
parallel pair:
compute both left and right-hand sides independently
return pair of values
Visualizing Computational Costs

(1 + 2 || f 3)

\[ (1 + 2) \]

\[ f(3) \]

\[ (, ,) \]
Visualizing Computational Costs

Suppose we have 1 processor. How much time does this computation take?

\[(1 + 2 \mid \mid f 3)\]
Visualizing Computational Costs

Suppose we have 1 processor. How much time does this computation take? Scheduling 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? Schedule A-CB-D: \(1 + \max(1, 7) + 1 = 9\)
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: \( 1 + \max(1, 7) + 1 = 9 \)
• 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 of 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)
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
- two series-parallel graphs in sequence
- two series-parallel graphs in parallel
Not a Series-Parallel Graph
Work and Span of Series-Parallel Graphs

Let's assume each node costs 1.

**Work**: sum the nodes.

**Span**: longest path from source to sink.
Let's assume each node costs 1.

**Work**: sum the nodes.

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

work = 10
span = 5
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
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  
|   E J  
|   J   
|   F   
|   F   

Diagram:
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
J F
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):
  - $T(p) < \frac{\text{work}}{p} + \text{span}$
  - $T(p) \geq \max(\frac{\text{work}}{p}, \text{span})$
    - can't do better than perfect division between processors ($\frac{\text{work}}{p}$)
    - can't be faster than span
Greedy Schedulers

Properties (for $p$ processors):

- $T(p) < \frac{\text{work}}{p} + \text{span}$

- $T(p) \geq \max(\frac{\text{work}}{p}, \text{span})$

Consequences:

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

- if span approaches the work
  - $\frac{\text{work}}{p} + \text{span} \Rightarrow \text{span}$
  - $\max(\frac{\text{work}}{p}, \text{span}) \Rightarrow \text{span}$
  - 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

Diagram:
- Split
- Recurse in parallel
- Merge
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
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) = \texttt{split} l in
    let (sorted1, sorted2) =
      \texttt{both} mergesort pile1
      mergesort pile2
    in
    \texttt{merge} sorted1 sorted2

Assume input list of size $n$:
\[
\text{work}_\text{mergesort}(n) = \text{work}_\text{split}(n) \\
+ 2 \cdot \text{work}_\text{mergesort}(n/2) \\
+ \text{work}_\text{merge}(n)
\]

\[
= k_1 n \\
+ 2 \cdot \text{work}_\text{mergesort}(n/2) \\
+ k_2 n
\]

\textit{read this as "approximately equal to"}
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) = k*n
+ 2*work_mergesort(n/2)
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)
                   = k*n             
                   + 2*work_mergesort(n/2)
                   = O(n log 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_splitsort(n) = span_split(n)
+ max(span_splitsort(n/2), span_splitsort(n/2))
+ span_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 (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) = k \times n  
\quad + k \times (n/2 + n/4 + n/8 + \ldots)
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 $n$:
\[ \text{span}_{\text{mergesort}}(n) = 2^k n \]
\[ = O(n) \]
let rec mergesort (l : int list) : int list =
    match l with
    []  -> []
    | [x] -> [x]
    | _   ->
        let (pile1,pile2) = \text{split} l in
        let (sorted1,sorted2) =
            \text{both} mergesort pile1
            mergesort pile2
        in
        \text{merge} sorted1 sorted2

Summary for input list of size n:
work\_mergesort(n) = k*n*log n
span\_mergesort(n) = k*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

Summary for input list of size $n$:
work$_{mergesort}(n) = k \cdot n \cdot \log n$  
span$_{mergesort}(n) = k \cdot n$

parallelism?  
parallelism = work/span  
= $n \cdot \log n / n$  
= $\log n$

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) = $k \times n \times \log n$

span\_mergesort(n) = $k \times n$

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

**parallelism?**

parallelism = work/span

= $n \times \log n / n$

= $\log n$

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.
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))

We are going to ignore this.
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
    in
tsort r
    merge (merge l' r') (one i)
Merging trees

- Sub-problem: 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 `split_at t i`
  - divides t in to items less than i and items greater than i

```plaintext
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')
```
Splitting a tree

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

```ml
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)
```

\[
\text{span } (h) = k \times h
\]

where \( h \) is the height of the tree \( t \)

\[
h = \log(n) \text{ if } t \text{ is balanced with } n \text{ nodes}
\]
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:

\[
\text{span\_merge}(h1,h2) = \text{span\_split}(h2) + \max(\text{span\_merge}(h1-1), \text{span\_merge}(h2-1)) = k \cdot h2 + \text{span\_merge}(h1-1) = k \cdot h2 \cdot h1
\]
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

\[
\text{span}_\text{tsort}(h) = \max(\text{span}_\text{tsort}(h-1), \text{span}_\text{tsort}(h-1)) + \text{span}_\text{merge}(h-1,h-1) + \text{span}_\text{merge}(h,1)
\]
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
Span of Parallel TreeSort

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

\[
\text{span}_\text{tsort}(h) = \max(\text{span}_\text{tsort}(h-1), \text{span}_\text{tsort}(h-1)) + \text{span}_\text{merge}(h-1,h-1) + \text{span}_\text{merge}(h,1) = \text{span}_\text{tsort}(h-1) + k*(h-1)*(h-1) + k*h
\]
\[
= k*h^3
= O(\log^3 n)
\]
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: $\text{work/span} = n \log n / \log^3 n = n / \log^2 n$
  - make use of millions of machines when sorting 10 billion items
  - caveat: we didn't factor in data communication costs!
Overall Summary

• **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
    • \(\text{Work}( e1 \ |\ | \ e2 ) = \text{Work}(e1) + \text{Work}(e2) + 1\)
  
  – *Span*: length of the longest dependency chain
    • time program execution takes with infinite processors
    • \(\text{Span}( e1 \ |\ | \ e2 ) = \max(\text{Span}\ e1, \text{Span}\ e2) + 1\)

  – **Parallelism**: \(\text{Work} / \text{Span}\)

• Many parallel algorithms follow a divide-and-conquer strategy
  
  – efficient algorithms divide quickly and merge quickly
END