today

parallel programming

- cost semantics
- Brent’s Theorem and speed-ups
- sequences: an abstract type with efficient parallel operations
parallelism

exploiting *multiple* processors
evaluating *independent* code *simultaneously*

- low-level implementation
  - *scheduling* work onto processors
- high-level planning
  - designing code *abstractly*
  - without *baking in a schedule*
our approach

design abstractly

- behavioral correctness
- asymptotic runtime (work, span)

reason abstractly

- independently of schedule
- cost semantics and evaluation
functional benefits

• No side effects, so…
  evaluation order doesn’t affect correctness

• Can build abstract types that support efficient parallel-friendly operations

• Can use work and span to predict potential for parallel speed-up
  • Work and span are independent of scheduling details
caveat

• In practice, it’s hard to achieve speed-up

• Current language implementations don’t make it easy

• Problems include:
  • scheduling overhead
  • locality of data (cache problems)
  • runtime sensitive to scheduling choices
why bother?

• It’s good to think *abstractly* first and figure out details later

  • Focus on *data dependencies* when you design your code

• Our thesis: this approach to parallelism will *prevail*...

  (and 15-210 builds on these ideas...)
cost semantics

We already introduced work and span

- **Work** estimates the *sequential* evaluation time on a *single* processor

- **Span** takes account of data dependency, estimates the *parallel* evaluation time with *unlimited* processors
cost semantics

• We showed how to calculate work and span for recursive functions with recurrence relations.

• Now we introduce cost graphs, another way to deal with work and span.

• Cost graphs also allow us to talk about schedules...

  ... and the potential for speed-up.
cost graphs

A cost graph is a **series-parallel graph**

- a *directed* graph, with *source* and *sink*
- nodes represent *units of work* (constant time)
- edges represent *data dependencies*
- branching indicates *potential parallelism*
series-parallel graphs

a single node

sequential composition

parallel composition
work and span

of a cost graph

- The **work** is the *number of nodes*
- The **span** is the *length of the longest path from source to sink*

\[ \text{span}(G) \leq \text{work}(G) \]
work \quad = \quad \text{work } G_1 \quad + \quad \text{work } G_2 \quad + \quad c

\text{dependent code … add the work}

work \quad = \quad \text{work } G_1 \quad + \quad \text{work } G_2 \quad + \quad c

\text{independent code … add the work}
\[ \text{span} = \text{span } G_1 + \text{span } G_2 + c \]

**dependent code ... add the span**

\[ \text{span} = \max(\text{span } G_1, \text{span } G_2) + c \]

**independent code ... max the span**
sources and sinks

- Sometimes we omit them from pictures
- No loss of generality
  - easy to put them in
- No difference, asymptotically
  - a single node represents an additive constant amount of work and span
- Allows easier explanation of execution
example

① and ② must be done before ⑦

work = 11 (number of nodes)
span = 4 (longest path length)

each node represents a single unit of work
using cost graphs

• Every expression can be given a cost graph
• Can calculate work and span using the graph
  • These are asymptotically the same as the work and span derived from recurrence relations

work and span provide asymptotic estimates of actual running time, under certain assumptions

work: single processor
span: unlimited processors

basic ops take constant time
scheduling

- Work: number of nodes
- Span: length of critical path

\[ w = 11 \]
\[ s = 4 \]

uses 5 processors

(i) 1 2 6 3 4
(ii) 7 5
(iii) 9 8
(iv) 10
(v) 11

an optimal parallel schedule
(5 rounds, or 4 steps)
What if there are only 2 processors?

2 processors cannot do the job as fast as 5 (!)
Brent's Theorem

An expression with work $w$ and span $s$ can be evaluated on a $p$-processor machine in time $O(\text{max}(w/p, s))$.

Optimal schedule using $p$ processors:
Do (up to) $p$ units of work each round
Total work to do is $w$
Needs at least $s$ steps

Richard Brent is an illustrious Australian mathematician and computer scientist. He is known for Brent's Theorem, which shows that a parallel algorithm can always be adapted to run on fewer processors with only the obvious time penalty—a beautiful example of an “obvious” but non-trivial theorem.
3 processors can do the work as fast as 5(!)

\[
\text{min } \{ p \mid \frac{w}{p} \leq s \} \text{ is 3}
\]

(a best schedule for 3 processors)

(5 rounds, 4 steps)
next

• Exploiting parallelism in ML
• A signature for parallel collections
• Cost analysis of implementations
• Cost benefits of parallel algorithm design
sequences

signature SEQ =

sig

  type 'a seq
  exception Range
  val tabulate : (int -> 'a) -> int -> 'a seq
  val length : 'a seq -> int
  val nth : int -> 'a seq -> 'a
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
end
implementations

• Many ways to implement the signature
  • lists, balanced trees, arrays, ...
• For each one, can give a cost analysis
• There may be implementation trade-offs
  • lists: item access is $O(n)$, length is $O(n)$
  • arrays: item access is $O(1)$, length is $O(1)$
  • trees: item access is $O(\log n)$, length is ??
An abstract parameterized type of sequences

Think of a sequence as a parallel collection

With parallel-friendly operations

constant-time access to items

efficient map and reduce

We’ll work today with an implementation

Seq : SEQ

based on vectors
sequence values

A value of type $t\ seq$ is a sequence of values of type $t$

- We use math notation like
  
  $\langle v_1, \ldots, v_n \rangle$
  
  $\langle v_0, \ldots, v_{n-1} \rangle$
  
  $\langle \rangle$

  for sequence values

$\langle 1, 2, 4, 8 \rangle$ is a value of type $int\ seq$
Two sequence values are (extensionally) equal iff they have the same length and have equal items at all positions

\[ \langle v_1, \ldots, v_n \rangle = \langle u_1, \ldots, u_m \rangle \]

if and only if

\[ n = m \] and for all \( i \), \( v_i = u_i \)
operations

For our given structure Seq : SEQ, we specify

- the (extensional) behavior
- the cost semantics

of each operation

Other implementations of SEQ are designed to have the same extensional behavior but may have different work/span profiles

Learn to choose wisely!
If $G_i$ is cost graph for $f(i)$, the cost graph for $\text{tabulate } f \ n$ is

\[ W(\text{tabulate } f \ n) = \sum \{ W(f \ 0), ..., W(f \ (n-1)) \} + c \]
\[ S(\text{tabulate } f \ n) = \max \{ S(f \ 0), ..., S(f \ (n-1)) \} + c \]

If $f$ is $O(1)$, the work for $\text{tabulate } f \ n$ is $O(n)$
If $f$ is $O(1)$, the span for $\text{tabulate } f \ n$ is $O(1)$
examples

• tabulate (fn x:int => x) 6  \langle 0, 1, 2, 3, 4, 5 \rangle
• tabulate (fn x:int => x*x) 6  \langle 0, 1, 4, 9, 16, 25 \rangle
• tabulate (fn _ => raise Range) 0  \langle \rangle
length

\[ \text{length} \langle v_1, ..., v_n \rangle = n \]

- Cost graph is
- Work is \( O(1) \)
- Span is \( O(1) \)

Contrast: \( \text{List.length } [v_1, ..., v_n] = n \)
work, span \( O(n) \)
nth

\[ \text{nth} \langle v_0, \ldots, v_{n-1} \rangle = v_i \]
\[ = \text{raise Range} \quad \text{otherwise} \]

- Work is \( O(1) \)
- Span is \( O(1) \)
- Cost graph is \( \text{Seq provides constant-time access to items} \)

Contrast with lists
examples

For \( n \geq 0 \), \( f \) total, \( 0 \leq i < n \),

\[
\text{length (tabulate } f \ n) = n
\]
\[
\text{nth } i \ (\text{tabulate } f \ n) = f \ i
\]

For all values \( S : t \ \text{seq} \),

\[
S = \text{tabulate } (\text{fn } i \ => \ \text{nth } i \ S) \ (\text{length } S)
\]
map

\[
\text{map } f \langle v_0, \ldots, v_{n-1} \rangle = \langle f v_0, \ldots, f v_{n-1} \rangle
\]

\[
\text{map } f \langle v_0, \ldots, v_{n-1} \rangle \text{ has cost graph}
\]

where each \( G_i \) is cost graph for \( f v_i \)

- If \( f \) is constant time, \( \text{map } f \langle v_0, \ldots, v_{n-1} \rangle \)
  has work \( O(n) \), span \( O(1) \)
  
  (contrast with List.map)
examples

map (fn x => x*x) ⟨1, 2, 3⟩ = ⟨1, 4, 9⟩

If \( f, g \) total then

\[
\text{map } g \ (\text{tabulate } f \ n) = \text{tabulate } (g \circ f) \ n
\]

map g ⟨f 0, ..., f(n-1)⟩

= ⟨g(f 0), ..., g(f(n-1))⟩

= ⟨(g \circ f) 0, ..., (g \circ f)(n-1)⟩
reduce

reduce should be used to combine a sequence using an associative function \( g \) with identity element \( z \)

- \( g : t \times t \rightarrow t \) is \textbf{associative} if for all \( x_1, x_2, x_3 : t \)
  \[
g(x_1, g(x_2, x_3)) = g(g(x_1, x_2), x_3)
  \]

- \( z \) is an \textit{identity element} for \( g \) if for all \( x : t \),
  \[
g(x, z) = x
  \]

We write \( v_0 \ g \ v_1 \ g \ldots \ g \ v_{n-1} \ g \ z \)
for the result of combining \( v_0, \ldots, v_{n-1}, z \)

\[
\text{reduce } g \ z \langle v_0, \ldots, v_{n-1} \rangle = v_0 \ g \ v_1 \ g \ldots \ g \ v_{n-1} \ g \ z \equiv v_0 \ g \ v_1 \ g \ldots \ g \ v_{n-1}
\]
examples

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Description</th>
<th>Identity Element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+)</td>
<td>int * int -&gt; int</td>
<td>associative, with identity element 0</td>
<td>0</td>
<td>( \text{reduce (op +) 0 } \langle v_0, ..., v_{n-1} \rangle = v_0 + ... + v_{n-1} )</td>
</tr>
<tr>
<td>(*)</td>
<td>int * int -&gt; int</td>
<td>associative, with identity element 1</td>
<td>1</td>
<td>( \text{reduce (op *) 1 } \langle v_0, ..., v_{n-1} \rangle = v_0 * ... * v_{n-1} )</td>
</tr>
<tr>
<td>(@)</td>
<td>t list * t list -&gt; t list</td>
<td>associative, with identity element []</td>
<td>[]</td>
<td>( \text{reduce (op @} [ ] } \langle v_0, ..., v_{n-1} \rangle = v_0 @ ... @ v_{n-1} )</td>
</tr>
</tbody>
</table>
reduce

• When g is associative and z is an identity

\[
\text{reduce } g \ z \ \langle v_0, \ldots, v_{n-1} \rangle = v_0 \ g \ v_1 \ g \ \ldots \ g \ v_{n-1} \ g \ z
\]

• If g is constant time,

\[
\text{reduce } g \ z \ \langle v_0, \ldots, v_{n-1} \rangle
\]

has work \(O(n)\) and span \(O(\log n)\)

(Contrast with foldr, foldl on lists)
reduce (op +) 0 ⟨1, 2, 3, 4, 5, 6, 7, 8⟩

cost graph
cost graphs

reduce splits the sequence into halves

\[
\text{reduce } g \ z \langle v_1, ..., v_{2n} \rangle = g(\text{reduce } g \ z \langle v_1, ..., v_n \rangle, \text{reduce } g \ z \langle v_{n+1}, ..., v_{2n} \rangle)
\]

Let \( W(m) \) = work for reduce \( g \ z \) \( S \) when length \( S = m \)

\[
W(2n) = 2 \times W(n) + c
\]

\[
S(2n) = S(n) + c
\]

\( W(n) \) is \( O(n) \)

\( S(n) \) is \( O(\log_2 n) \)
mapreduce

- When $g$ is associative and $z$ is an identity,
  $$\text{mapreduce } f \ z \ g \ \langle v_0, \ldots, v_{n-1} \rangle = (f \ v_0) \ g \ldots \ g \ (f \ v_{n-1}) \ g \ z$$

- When $f, g$ are constant time,
  $$\text{mapreduce } f \ z \ g \ \langle v_0, \ldots, v_{n-1} \rangle$$
  has work $O(n)$
  and span $O(\log n)$
fun sum (s : int seq) : int =
    reduce (op +) 0 s

fun count (s : int seq seq) : int =
    sum (map sum sum s)
Let $s$ be a value of type \texttt{int seq seq} consisting of $n$ rows, each of length $n$.

What are the work and span for $\text{count } s$?
Let \( s = \langle s_1, ..., s_n \rangle \), \( s_i = \langle x_{i1}, ..., x_{in} \rangle \), \( t_i = \text{sum } s_i \)

For each \( i \), \( \text{sum } s_i = \text{reduce}(\text{op } +) \ 0 \ \langle x_{i1}, ..., x_{in} \rangle \)

Cost graph of \( \text{sum } s_i \):

- \( \log_2 n \)
- Work is \( O(n) \)
- Span is \( O(\log n) \)

Map \( \text{sum } s = \langle \text{sum } s_1, ..., \text{sum } s_n \rangle \)

Cost graph of \( \text{map } \text{sum } s \):

- Work is \( O(n^2) \)
- Span is \( O(\log n) \)
Let $t_i = \text{sum } s_i$

\[
\text{count } s = \text{sum } \langle t_1, \ldots, t_n \rangle
\]

cost graph of $\text{sum (map sum s)}$

work is $O(n^2)$
span is $O(\log n)$
exercises

- Define functions

  ```
  reverse : 'a seq -> 'a seq
  zip : 'a seq * 'b seq -> ('a * 'b) seq
  ```

  and analyze their work and span
fun reverse (s : 'a seq) : 'a seq = 
  let
    val n = length s
  in
    tabulate (fn i => nth (n - i - 1) s) n
  end

What are the work and span for reverse \(<v_1, \ldots, v_n>\)?