today
Parallel evaluation, using *sequences*

---

**GRAVITY**

...*don't leave home without it!*
signature SEQ =

sig
  type 'a seq
  exception Range
  val nth : int -> 'a seq -> 'a
  val length : 'a seq -> int
  val tabulate : (int -> 'a) -> int -> 'a seq
  val empty : unit -> 'a seq
  val map : ('a -> 'b) -> ('a seq -> 'b seq)
  val split : 'a seq -> 'a seq * 'a seq
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
end
sequences

• The SEQ signature can be implemented in various different ways
  • lists, trees, arrays, vectors, …
  • balanced, sized, …

• Each implementation provides the ingredients in SEQ, with its own work/span characteristics
  • lists: \( \text{nth i S is } O(n) \)
  • balanced trees: \( \text{nth i S is } O(\log n) \)
  • arrays: \( \text{nth i S is } O(1) \)
your task

• Assume given a structure $\text{Seq : SEQ}$ with known work/span characteristics

• Design efficient and correct solutions to parallelizable problems
  • prove correctness
  • calculate work and span
We can also talk about our sequence operations *abstractly*, in a way that’s independent of the implementation.

Example:

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

Even though \text{map} may not have been defined this way, this equation is valid and both sides have the same \text{work/span}.
behavior

empty( ) = ⟨ ⟩

length ⟨v₀,…,vₙ₋₁⟩ = n

nth i ⟨v₀,…,vₙ₋₁⟩ = vᵢ if 0 ≤ i < n

tabulate f n = ⟨f(0), …, f(n-1)⟩

split ⟨v₀,…,vₙ₋₁⟩ = (⟨v₀,…,vₘ₋₁⟩, ⟨vₘ,…,vₙ₋₁⟩)

where m = n div 2
reduce

fun reduce g z s =
  case (length s) of
    0 => z
  | 1 => g(nth 0 s, z)
  | _ => let
    val (s1, s2) = split s
  in
    g(reduce g z s1, reduce g z s2)
end

reduce g z ⟨v₁,…,vₙ⟩ = v₁ g v₂ g … vₙ g z
when g is associative, z an identity for g
mapreduce

fun mapreduce f z g s =
  case (length s) of
    0 => z
  | 1 => g (f(nth 0 s), z)
  | _ => let
    val (s1, s2) = split s
    in
    g(mapreduce f z g s1, mapreduce f z g s2)
  end

mapreduce f g z ⟨v_1, ..., v_n⟩
  = (f v_1) g (f v_2) g ... (f v_n) g z
  when g is associative, z an identity for g
example

reduce \( (\text{op } +) \) 0 \( \langle v_1, v_2 \rangle \)

\[ = (\text{op } +) (\text{reduce } (\text{op } +) 0 \langle v_1 \rangle, \text{reduce } (\text{op } +) 0 \langle v_2 \rangle) \]

\[ = (\text{reduce } (\text{op } +) 0 \langle v_1 \rangle) + (\text{reduce } (\text{op } +) 0 \langle v_2 \rangle) \]

\[ = (v_1 + 0) + (v_2 + 0) \]

\[ = v_1 + v_2 \]

reduce \( g \ z \) behaves “correctly” when \( g \) is associative and \( z \) is an identity element

\[ \text{reduce } (\text{op } +) 0 \langle v_1, v_2 \rangle = v_1 + v_2 + 0 \]
example

\[
\text{reduce (op +) } 21 \langle v_1, v_2 \rangle \\
= (\text{op +) (reduce (op +) } 21 \langle v_1 \rangle, \text{reduce (op +) } 21 \langle v_2 \rangle) \\
= (\text{reduce (op +) } 21 \langle v_1 \rangle) + (\text{reduce (op +) } 21 \langle v_2 \rangle) \\
= (v_1 + 21) + (v_2 + 21) \\
= v_1 + v_2 + 42
\]

reduce (op +) 21 \langle v_1, v_2 \rangle \neq v_1 + v_2 + 21
thinking abstractly

• Use **cost semantics** to predict work and span of code

  • *before* testing or *correctness* analysis

• Use **behavioral specs** to guide us to design correct code

• Use **inductive proof** methods to *validate* specs and *confirm* cost analysis
gravitation

• Newtonian laws

• Simulate the motion of planets
  • for n bodies, this is $O(n^2)$ work

• Using sequences and parallel operations is very natural (!)
  • and faster than using lists
Newton’s laws

\[ F = G \frac{m_1 \cdot m_2}{r^2} \]

- **Point masses** attract each other with a **force** proportional to the **product** of the masses and the **inverse square** of the distance.

- **Spherical bodies** behave like point masses.
laws of motion

Law 1: If an object experiences no net force, its velocity is constant:
- it moves in a straight line, with constant speed.

Law 2: The acceleration of a body is parallel and proportional to the net force acting on the body, and inversely proportional to the mass of the body, i.e., \( F = m \ a \).

Law 3: When one body exerts a force \( F \) on a second body, the second body exerts an equal but opposite force \( -F \) on the first.

Law 4: There is no Law 4.
Velocity, force and acceleration are **vectors**

- Vectors have **magnitude** and **direction**
  
  \[
  \text{speed} = \text{magnitude of velocity}
  \]

- Vectors can be **added**

  \[
  \text{velocity} + \text{velocity} = \text{velocity} \\
  \text{acceleration} + \text{acceleration} = \text{acceleration}
  \]

- Vectors can be **multiplied** by a scalar

  \[
  \text{scalar} \times \text{velocity} = \text{velocity} \\
  \text{scalar} \times \text{acceleration} = \text{acceleration}
  \]
our version

• 2-dimensional universe

• Scalars are real numbers

• Vectors are pairs of type real * real

Easy to generalize...
bodies

• A body has position, mass, and velocity
• Positions are points, pairs of real numbers
• A mass is a (positive) real number
• A velocity is a 2D-vector
  • also represented as a pair of reals

\[
\begin{align*}
\textbf{type} & \quad \text{point} = \text{real} \times \text{real} \\
\textbf{type} & \quad \text{vect} = \text{real} \times \text{real} \\
\textbf{type} & \quad \text{body} = \text{point} \times \text{real} \times \text{vect}
\end{align*}
\]
vectors

signature VECT =
  sig
    type vect = real * real
    val zero : vect
    val add : vect * vect -> vect
    val scale : real * vect -> vect
    val mag : vect -> real
  ...
end
structure Vect : VECT =
struct
  type vect = real * real

  val zero = (0.0, 0.0)

  fun add ((x1, y1), (x2, y2)) = (x1+x2 , y1+y2)

  fun scale(c, (x,y)) = (c * x , c * y)

  fun mag (x,y) = Math.sqrt (x * x + y * y)
end
points

type point = real * real

fun diff ((x1,y1):point, (x2,y2):point) : vect
  = (x2 - x1, y2 - y1)

fun displace ((x,y):point, (x',y'):vect) : point
  = (x + x', y + y')
bodies

(  position,  mass,  velocity  )

type body = point * real * vect

val sun = ((0.0,0.0), 332000.0, (0.0,0.0))

val earth = ((1.0, 0.0), 1.0, (0.0, 18.0))

distance from sun to earth
  = one "astronomical unit"

sun is 332000 times more massive

the sun’s (relative) velocity is zero
motion

- To calculate the *motion* of a body in a *timestep*
  - find the net *acceleration* due to other bodies
  - adjust the *position* and *velocity* of the body
accel

accel : body -> body -> vect

accel b₁ b₂ = acceleration on b₁
due to gravitational attraction of b₂

use default of zero
when bodies are too close

fun accel (p₁, _, _) (p₂, m₂, _) =
    let
        val d = diff(p₁, p₂)
        val r = mag d
    in
        if r < 0.1 then zero else scale(G * m₂/(r*r*r) , d)
    end
\[ r = \text{distance from } p_1 \text{ to } p_2 \]

\[ \text{accel} = \frac{G m_2}{r^2} \]

\[ \text{due to } b_2 \]

\[ \text{= acceleration on } b_1 \]
accel

\[ r = \text{distance from } p_1 \text{ to } p_2 \]

= acceleration on \( b_2 \) due to \( b_1 \)
accels

accels : body -> body seq -> vect

\[\text{accels } b \ s = \text{net acceleration on } b\]

\[\text{due to gravitational attraction}\]

\[\text{of the bodies in } s\]

\textbf{fun} accels b s =

mapreduce (accel b) zero add s

\[\text{accels } b \ 〈b₁,\ldots,bₙ〉 =\]

\[\text{accel } b \ b₁ + \ldots + \text{accel } b \ bₙ\]

(vector sum)
fun move (p, m, v) (a, dt) =
  let
    val dp = add(scale(dt,v), scale(0.5*dt*dt, a))
    val dv = scale(dt, a)
  in
    (add(p, dp), m, add(v, dv))
end

move (p, m, v) (a, dt) = (p', m, v')
  v' = v + a dt
  p' = p + v dt + 1/2 a dt^2

Newtonian calculus, too!
step

step : real -> body seq -> body seq

parallel evaluation
- each body calculates its own update

fun step dt s =
  map (fn b => move b (accels b s, dt)) s

step dt \langle b_1, b_2, ..., b_N \rangle = \langle b_1', b_2', ..., b_N' \rangle

where, for each i,
  \[ b'_i = \text{move} \ b_i \ (a_i, dt) \]
  and \[ a_i = \text{accels} \ b_i \ \langle b_1, b_2, ..., b_N \rangle \]
efficiency

- What are the **work** and **span** for

  \[ \text{accel } b_i \, b_j \]

  \[ \text{accels } b_i \, \langle b_1, \ldots, b_N \rangle \]

  \[ \text{move } b \, (a, dt) \]

  \[ \text{step } dt \, \langle b_1, \ldots, b_N \rangle \]
Assume we have an implementation of SEQ with

<table>
<thead>
<tr>
<th>expression</th>
<th>work</th>
<th>span</th>
</tr>
</thead>
<tbody>
<tr>
<td>nth i s</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>length s</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>tabulate f n</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>empty( )</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>map f s</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>reduce g z s</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>mapreduce f g z s</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>split s</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

when length of $s$ is $n$, and $f, g$ are constant time
accel \( (p_1, m_1, v_1) \) \( (p_2, m_2, v_2) = \)

\[
\text{let}
\begin{align*}
\text{val } d &= \text{diff}(p_1, p_2) \\
\text{val } r &= \text{mag } d
\end{align*}
\text{in}
\begin{align*}
\text{if } r < 0.1 & \text{ then zero else } \text{scale}(G \cdot m_2/(r^3), d)
\end{align*}
\text{end}
\]

work, span \( O(1) \)

accel \( b_1 \) \( b_2 \) has

work \( O(1) \)

span \( O(1) \)
accels

\[
\text{accels } b_i \left< b_1, \ldots, b_N \right> = \text{mapreduce (accel } b \text{) zero add } \left< b_1, \ldots, b_N \right>
\]

\[
\text{mapreduce } f \ z \ g \left< b_1, \ldots, b_N \right> \text{ applies } f \text{ N times in parallel and combines using } g
\]

\[
\text{accels } b_i \left< b_1, \ldots, b_N \right> \text{ has work } O(N), \text{ span } O(\log N)
\]
move (p, m, v) (a, dt) =
    let
      val p' = displace(p, add(scale(dt,v), scale(0.5*dt*dt, a)))
      val v' = add(v, scale(dt, a))
    in (p', m, v')
end

work, span O(1)
Let $s$ be $\langle b_1, ..., b_N \rangle$.

$$\text{step } dt \ s = \ \text{map } (\text{fn } b \Rightarrow \text{move } b \ (\text{accels } b \ s, dt)) \ s$$

$\text{step } dt \ \langle b_1, ..., b_N \rangle$ has work $O(N^2)$, span $O(\log N)$.
## Cost Analysis

(Using sequences)

<table>
<thead>
<tr>
<th></th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{accel } b_i b_j$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$\text{accels } b_i \langle b_1, \ldots, b_N \rangle$</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>$\text{move } b (a, dt)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$\text{step } dt \langle b_1, \ldots, b_N \rangle$</td>
<td>$O(N^2)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>
**cost analysis**
(Using lists)

\[
\text{fun accels } b \ (L : \text{body list}) = \\
\quad \text{foldr add zero (List.map (accel b) L)}
\]

\[
\text{fun step dt } (L : \text{body list}) = \\
\quad \text{List.map (fn } b \Rightarrow \text{move } b \ (\text{accels } b \ L, dt) \ ) \ L
\]

<table>
<thead>
<tr>
<th></th>
<th>work</th>
<th>span</th>
</tr>
</thead>
<tbody>
<tr>
<td>accels } b_i \ \langle b_1, \ldots, b_N \rangle</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>step dt \ \langle b_1, \ldots, b_N \rangle</td>
<td>O(N^2)</td>
<td>O(N^2)</td>
</tr>
</tbody>
</table>
conclusion

- Sequences allow efficient exploitation of parallel evaluation
  - $O(\log N)$ is better than $O(N)$
- In practice, can deliver real speed-up
- But there’s still room for improvement...
mini-solar system

val sun = ((0.0,0.0), 332000.0, (0.0,0.0))
val earth = ((1.0, 0.0), 1.0, (0.0,18.0))

us = ⟨sun, earth⟩

val us : body seq =
  tabulate (fn 0 => sun | 1 => earth | _ => raise Range) 2
  step us 0.01
  =>* ⟨((5E~05,0.0),332000.0,(0.01,0.0)),
        ((~15.6,0.18),1.0,(~3320.0,18.0))⟩
fun orbit b (n, dt) = 
  if n=0 then [ ] else 
    let 
      val (p', m, v') = move b (accel b sun, dt) 
    in 
      p' :: orbit (p', m, v') (n-1, dt) 
  end;
results

orbit earth (10, 0.01) =

[ (~15.6, 0.18), (~48.7318019171, 0.359213099043),
 (~81.7884162248, 0.537587775754),
 (~114.835559608, 0.71589462109),
 (~147.878962872, 0.894177309445),
 (~180.920348361, 1.07244756107),
 (~213.960467679, 1.25071021688),
 (~246.999717285, 1.42896774708),
 (~280.03833222, 1.60722158368),
 (~313.076463412, 1.78547263142)]
shaped like an ellipse