15-150 Fall 2012
Lecture 20

Stephen Brookes

today

Parallel evaluation, using ***sequences***

- work & span analysis

sequences

```
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 reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
end
```

specifications

work and span

gravitation

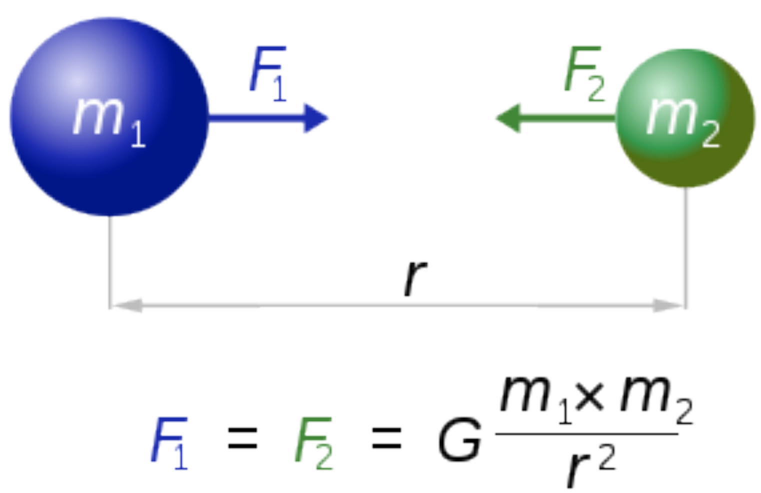
- Newtonian laws of motion
- Newton's gravitational law
- Simulate the motion of stars, planets, etc
- For n bodies, requires $O(n^2)$ work
- Using sequences we can improve the *span*
  - use of parallel evaluation is *natural*

gravity law

$F = G\, m_1\, m_2 / r^2$

- A *point mass* attracts another *point mass* with a ***force*** proportional to the *product* of their masses and the *inverse square* of their distance, directed along the line connecting them
- *Spherical bodies* behave like point masses with all the mass at their center

Newton, 1687



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 **a** of a body is parallel and proportional to the net force **F** acting on the body, and inversely proportional to the mass *m* of the body, i.e., **F** = *m* **a**.

Law 3: When a first body exerts a force **F** on a second body, the second body exerts an equal but opposite force –**F** on the first body.

vectors

- Velocity, force and acceleration are ***vectors***
- Vectors have ***magnitude*** and ***direction***

  speed = *magnitude* of velocity
- Vectors can be ***added***

  velocity + *velocity* = *velocity*

  acceleration + *acceleration* = *acceleration*
- Vectors can be ***multiplied*** by a *scalar*

  scalar * *velocity* = *velocity*

  scalar * *acceleration* = *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 vector

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

vectors

```
type vect = real * real

val zero : vect = (0.0, 0.0)

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

fun scale(c:real, (x,y):vect):vect
            = (c * x , c * y)

fun mag ((x,y) : vect) : real
            = Math.sqrt (x * x + y * y)
```

Euclidean

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

let's face it, that sun isn't going anywhere fast!

accel

accel : body -> body -> vect

```
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
```

accel b_1 b_2 =
acceleration on b_1
due to gravity of b_2

accels

accels : body -> body seq -> vect

```
fun accels b s =
  mapreduce (accel b) zero add s
```

accels b $\langle b_1,\ldots,b_n\rangle$ =
accel b b_1 + ... + accel b b_n

net acceleration on b
due to gravitational attraction
of the bodies in s

move

move : body -> vect * real -> body

```
fun 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
```

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!

move

move (p, m, v) (a, dt) = (p′, m, v′)

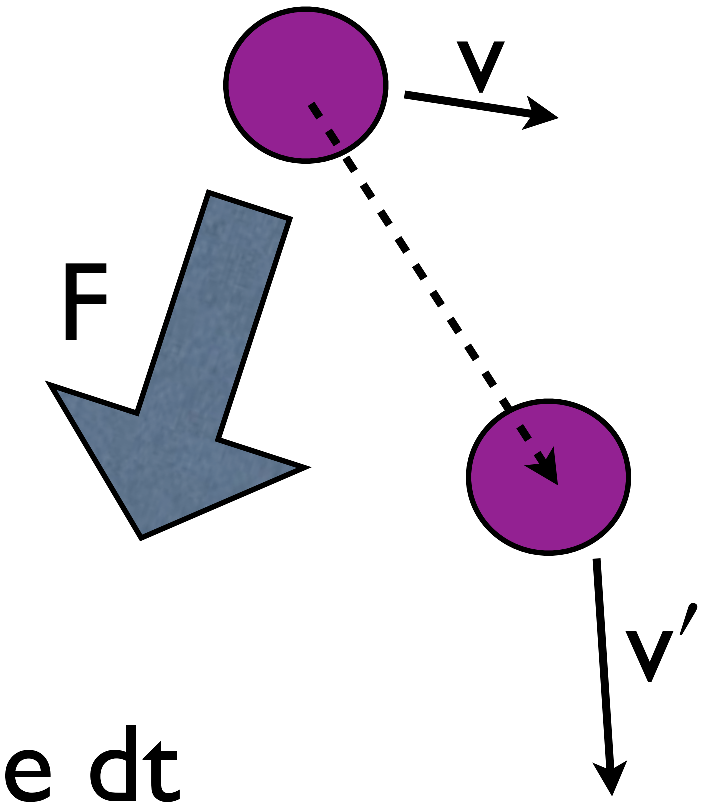
when

body at p, mass m, velocity v

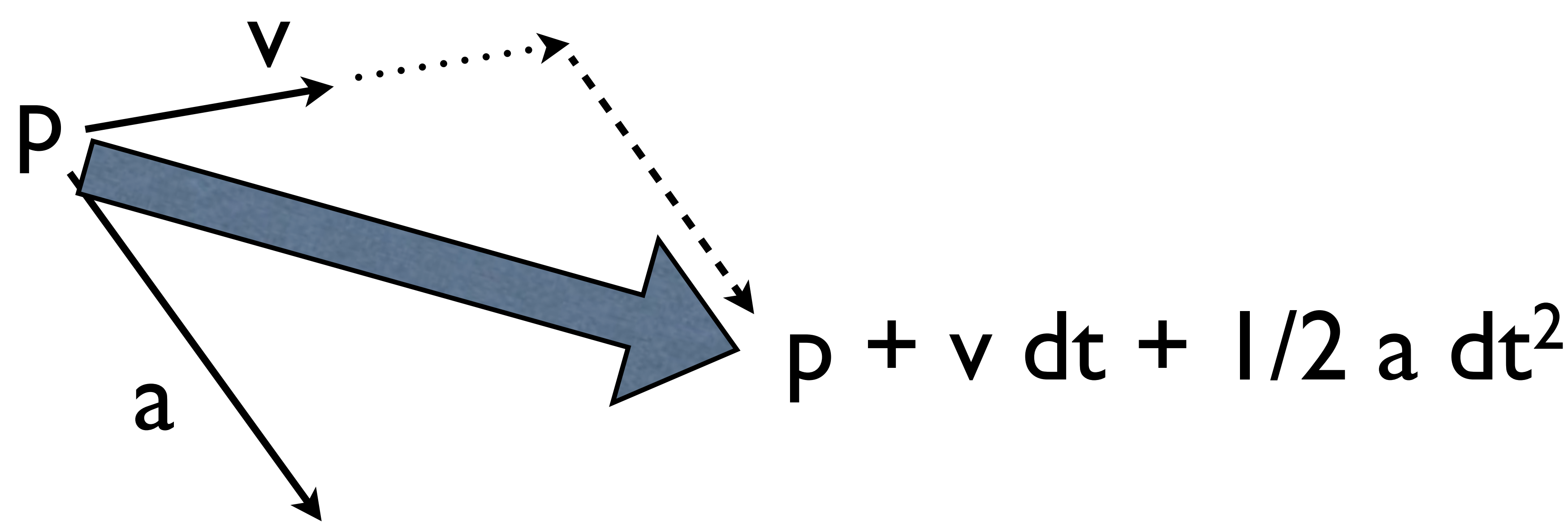
acted on by force F = m * a for time dt

moves to p′

and its velocity changes to v′



updating position



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

accel b_i b_j

accels b_i $\langle b_1, ..., b_N \rangle$

move b (a, dt)

step dt $\langle b_1, ..., b_N \rangle$

?

accel

```
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
```

work, span O(1)

accel b_1 b_2 has
work O(1)
span O(1)

accels

work, span $O(1)$

$\text{accels } b_i \langle b_1, \ldots, b_N \rangle =$

$\quad \text{mapreduce (accel b) zero add } \langle b_1, \ldots, b_N \rangle$

$\text{mapreduce f z g } \langle b_1, \ldots, b_N \rangle$
applies f N times in parallel
and combines using g

$f\, b_1 \quad f\, b_2 \; \ldots \quad f\, b_N$

$g \; \ldots \qquad g$

g

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

has work $O(N)$, span $O(\log N)$

move

work, span O(1)

```
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)

```
move (p, m, v) (a, dt)
```

has work, span O(1)

step

Let s be $\langle b_1, \ldots, b_N \rangle$

work O(N), span O(log N)

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

map f s   calls f, N times

step dt $\langle b_1, \ldots, b_N \rangle$
has work O(N*N),
span O(log N)

N sequential calls

N parallel calls

cost analysis

	work	span
accel b_i b_j	O(1)	O(1)
accels b_i $\langle b_1, ..., b_N \rangle$	O(N)	O(log N)
move b (a, dt)	O(1)	O(1)
step dt $\langle b_1, ..., b_N \rangle$	O(N^2)	O(log N)

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))⟩

orbit

orbit : body -> int * real -> point list

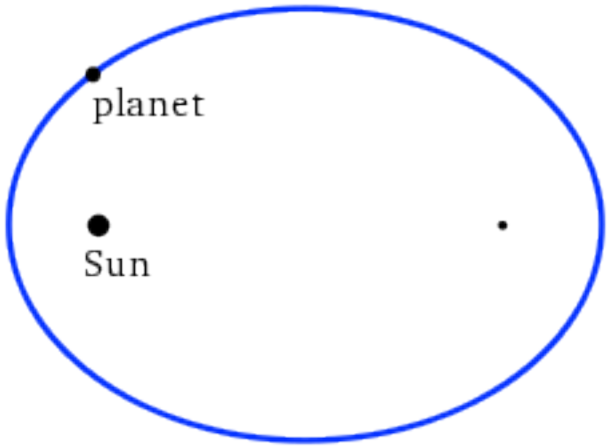
orbit b (n, dt) = first n positions of b in orbit around sun

```
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;
```

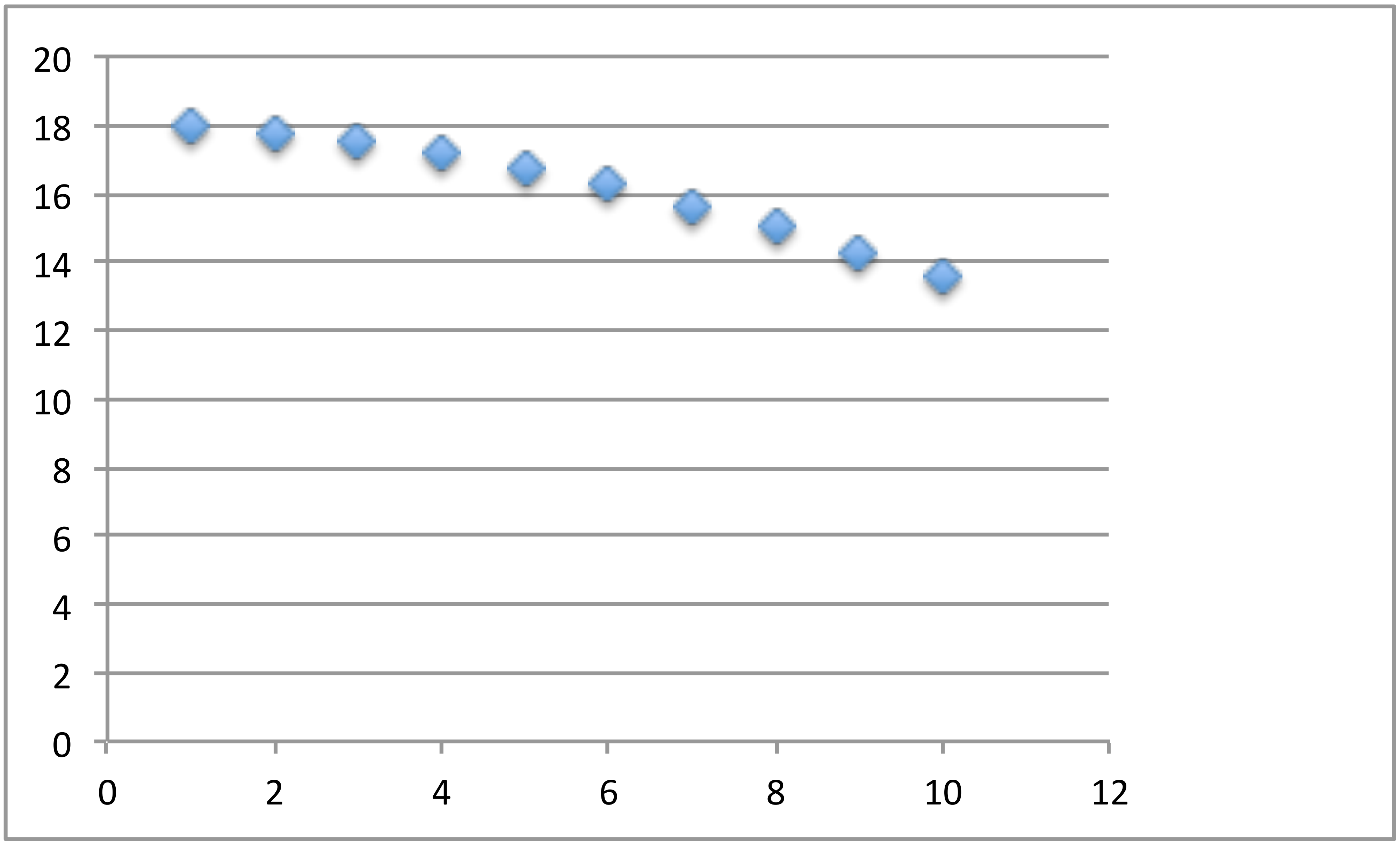
results

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),
(~313.07646341 2,1.78547263142)]



results



orbit of earth

cost analysis

(using lists)

```
fun accels b (L:body list) =
      List.foldr add zero (List.map (accel b) L)

fun step dt (L:body list) =
      List.map (fn b => move b (accels b s, dt)) s
```

	work	span
accels b_i $\langle b_1, ..., b_N\rangle$	O(N)	O(N)
step $\langle b_1, ..., b_N\rangle$ dt	$O(N^2)$	$O(N^2)$

cost analysis

(using sequences)

```
fun accels b (L:body Seq.seq) =
      foldr add zero (Seq.map (accel b) L)

fun step dt (L:body Seq.seq) =
      Seq.map (fn b => move b (accels b s, dt)) s
```

	work	span
accels b_i $\langle b_1, ..., b_N \rangle$	$O(N)$	$O(\log N)$
step $\langle b_1, ..., b_N \rangle$ dt	$O(N^2)$	$O(N \log N)$

conclusion

- Using sequences allows us to exploit the potential for parallel evaluation
- O(log N)   is better than O(N)
- In practice, can deliver real speed-up
- But there's still room for improvement...