15-150 Lectures 17 and 18:
Sequences, n-Body Simulation

Lectures by Dan Licata

March 20 and 22, 2012

Before we start, let's think about the big picture of parallelism. Parallelism is relevant to situations where many things can be done at once—e.g. using the multiple cores in multi-processor machine, or the many machines in a cluster. Overall, the goal of parallel programming is to describe computation in such a way that it can make use of this ability to do work on multiple processors simultaneously. At the lowest level, this means deciding, at each moment in time, what to do on each processor. This is limited by the data dependencies in a problem or a program. For example, evaluating `(1 + 2) + (3 + 4)` takes three units of work, one for each addition, but you cannot do the outer addition until you have done the inner two. So even with three processors, you cannot perform the calculation in fewer than two timesteps. That is, the expression has work 3 but span 2.

Now, one way to do parallel programming is to explicitly say what to do on each processor at each timestep—this is called a *schedule*. There are languages that let you write out a schedule explicitly, but there are disadvantages: For example, when you buy a new machine, you need to adapt your program that was written for 4 processors to 16, or 64, or a one million... Moreover, it's tedious and boring to think about assigning work to processors, when what you want to be thinking about is the problem you're trying to solve.

The approach to parallelism that we're advocating in this class is based on raising the level of abstraction at which you can think, by *separating the specification of what work there is to be done from the schedule that maps it onto processors.* As much as possible, you, the programmer, worry about specifying what work there is to do, and the compiler takes care of scheduling it onto processors. Three things are necessary to make this separation of concerns work:

1. The code itself must not bake in a schedule.

2. You must be able to reason about the *behavior* of your code independently of the schedule.

3. You must be able to reason about the *time complexity* of your code independently of the schedule.

Our central tool for avoiding baking in a schedule is *functional programming.* First, we focus on bulk operations on big collections which do not specify a particular order in which the operations on each element are performed. For example, today, we will talk about *sequences*, which come with an operation `map f <x1,x2,...,xn>` that is specified by saying that its value is the sequence `<f x1, f x2, ... f xn>`. This specifies the data dependencies (to calculate `map`, you need to calculate `f x1` ...) without specifying a particular schedule. You can implement the same computation with a loop, saying "do `f x1`, then do `f x2`,...", but this is inlining a particular schedule into the code—which is bad, because it gratuitously throws away opportunities for parallelism.

Second, functional programming focuses on pure, mathematical functions, which are evaluated by calculation. This limits the dependence of one chunk of work on another to what it is obvious from the data-flow in the program. For example, when you `map` a function `f` across a sequence, evaluating `f` on the first element has no influence on the value of `f` on the second element, etc.—this is not the case for imperative programming, where one call to `f` might influence another via memory updates. It is in general undecidable to take an imperative program and notice notice, after the fact, that what you really meant by that loop was a bulk operation on a collection, or that this particular piece of code really defines a mathematical function.

Our central tool for reasoning about behavior, independently of the schedule, is *deterministic parallelism*: we ensure that the behavior of your program is in fact *the same* for any schedule! The is true because programs have a well-defined mathematical meaning independent of any implementation. This meaning is specified by the *evaluation semantics.* For example, we say that

```
map f <x1,...,xn> == <f x1, ... , f xn>
```

We don't say exactly how this steps, because there are many possible stepping strategies than have this same evaluation behavior. We might evaluate sequentially, only running one `f xi` at a time, or in parallel, allowing them all to step in one timestep.

Our central tool for reasoning about time complexity, independently of the schedule, is a *cost semantics.* This includes the asymptotic work/span analyses that we have been doing all semester. The cost semantics lets you reason abstractly, and constrains how the scheduler divides up the work. For example, the cost semantics for `map` will say that `map (fn x => x + 1) s` takes $O(n)$ work but $O(1)$ span—each function application can be done in parallel. This constrains the implementation of `map` to one that makes this bound come true.

However, there's an important caveat, which is that given today's technology, this separation—you worry about saying what there is to do, the compiler schedules it—is not always a practical way to get good performance. Getting parallel programs to run quickly is hard, and there are lots of smart researchers actively working on it. Some of the issues include: overhead (it takes to distribute tasks to processors, notice when they've been completed, etc.); spatial locality (how do you make sure the necessary data can be accessed quickly?); schedule-dependence (sometimes, the schedule can make an asymptotic difference in time or space usage). So I don't want you to get the impression that writing programs the way we're asking you to write them will magically guarantee good performance. There are some implementations of functional languages that address these issues, to varying degrees: Manticore, MultiMLton, and PolyML are implementations of Standard ML that has a multi-threaded run-time, so you *can* run the sequence code that we will write in parallel. But it's tricky to get actual speedups. NESL is a research language by Guy Blelloch (who designed 15-210), and is where a lot of the ideas that we will discuss today originated; there is real implementation of NESL and some benchmarks with actual speedups on multiple processors. GHC, a Haskell compiler, implements many of the same ideas, and you can get some real speedups there too. Scala is a hybrid functional/object-oriented language, and one of the TAs implemented sequences in Scala and has gotten some performance gains on the 210 assignments.

So why are we teaching you this style of parallel programming? There are two reasons: First, even if you have to get into more of the gritty details of scheduling to get your code to run fast today, it's good to be able to think about problems at a high level first, and then figure out the details. If you're writing some code for an internship this summer using a low-level parallelism interface, it can be useful to first think about the abstract algorithm—what are the dependencies between tasks? what can possibly be done in parallel?—and then figure out the details. You can use parallel functional programming to design algorithms, and then translate them down to whatever interface you need. Second, it's our thesis that eventually this kind of parallel programming will be practical and common: as language implementations improve, and computers get more and more cores, this kind of programming will become possible and even necessary. You're going to be writing programs for a long time, and we're trying to teach you tools that will be useful years down the road.

The Plan In these two lectures, we are going to discuss two things: We will discuss *sequences*, an important data structure with good parallel complexity. And we will use these tools to do n-body simulation.

One guiding principle is

Theorem 1 (Brent's Theorem). *An expression with work w and span s can be run on a p-processor machine in time $O(max(w/p, s))$.*

That is, you try to divide the total work w up into chunks of p, but you can never do better than the span s. For example, if you have 10 units of work to do and span 5, you can achieve the span on 2 processors. If you have 15 units of work, it will take at least 8 steps on 2 processors. But if you increase the number of processors to 3, you can achieve the span 5. If you have 5 units of work to do, the fact that you have 2 processors doesn't help: you still need 5 steps.

Brent's theorem should be true for any language you design; otherwise, you have gotten the cost semantics wrong! Thus, we will sometimes refer to it as *Brent's Principle*: a language should be designed so that Brent's Principle is in fact a theorem.

1 Sequences

In the first lecture, you acted out counting the number of students who took 122 last semester, and then we implemented this using sequences. Now you're in a position to understand exactly what's going on:

```
fun sum (s : int Seq.seq) : int = Seq.reduce (fn (x,y) => x + y) 0 s

(* count the number of students in the class who have taken 122 *)
fun count (students : int Seq.seq Seq.seq) : int =
    sum (Seq.map sum students)
```

The type `Seq.seq` represents sequences. Sequences are *parallel collections*: ordered collections of things, with parallelism-friendly operations on them. Don't think of sequences as being implemented by lists or trees (though you could implement them as such); think of them as a new built-in type with only the operations we're about to describe. The differences between sequences and lists or trees is the cost of the operations, which we specify below.

Returning to the code, `sum` takes an integer sequence and adds up all the numbers in it using `reduce`, just like you have seen with reduce for lists and trees. `count` sums up all the numbers in a sequence of sequences, by (1) summing each individual sequence and then (2) summing the sequence that results. (Exercise: rewrite `count` with `mapreduce`, so it takes only one pass). You understand `map` and `reduce` for lists and trees, and the operations for sequences do the analogous thing.

Here are some of the operations we will use:

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

Why do we write `Seq.seq`,`Seq.map`, etc.? The reason is that sequences are packaged up in a *module*, which we will talk about next week. For now, just think of the whole phrase `Seq.map` as the name of a function.

Intuitively, these sequence operations do the same thing as the operations on lists that you are familiar with. However, they have different time complexity than the list functions: First, sequences admit constant-time access to elements—`nth` takes constant time. Second, sequences have better parallel complexity—many operations, `map` act on each element of the sequence in parallel.

For each function, we (1) describe its behavior abstractly and (2) give a cost specification.

Nth Sequences provide constant-time access to elements. Abstractly, we define the behavior of `nth` as

```
nth <x0 , ... , xn-1> i == xi if 0 <= i < n
                              or raises Range if i>=n
```

Here `<x1 , ... , xn>` is *not* SML syntax, but mathematical syntax for a sequence value.

`nth` has $O(1)$ work/span.

Length The behavior of `length` is

```
length <x1 , ... , xn> == n
```

`length` has $O(1)$ work/span.

Map The the behavior of `map f <x1,...xn>` is

```
map f <x1 , ... , xn> ==> <f x1,...,f xn>
```

If `f` takes constant time, then `map f` has $O(n)$ work and $O(1)$ span. Otherwise, the work is the sum of the time to run `f` on each argument, and the span is the max of the time to run `f` on each argument.

Reduce The behavior of `reduce` is

$$\texttt{reduce}\odot\ b\,\langle x_1,\ldots x_n\rangle \cong x_1 \odot x_2 \odot \ldots \odot x_n$$

That is, `reduce` applied its argument function (which we write here as infix $\odot$) between every pair of elements in the sequence.

However, the right-hand side is ambiguous, because we have not parenthesized it. There are two options: First, we could assume the function $\odot$ is associative, with unit b, in which case these all mean the same thing. However, there are some useful non-associative operations (e.g. floating point). So the second option is to specify a particular parenthesization $(x_1 \odot (x_2 \odot \ldots \odot (x_n \odot b)) \ldots)$ or $(\ldots (x_1 \odot x_2) \odot \ldots \odot x_n)$ or the balanced tree $((x_1 \odot x_2) \odot (x_3 \odot x_4)) \odot \ldots$ (with b filled in at the end if the sequence has odd length). Unless we say otherwise, you can assume the balanced tree.

If $\odot$ takes constant time (e.g. `op+`) then the graph has size (work) $O(n)$ and critical path length (span) $O(\log n)$. Reduce does not have constant span, because later applications of $\odot$ depend on the values of earlier ones.

Example We can now see that `count` on an $n \times n$ classroom has $O(n^2)$ work: `sum s` is implemented using `reduce` with constant-time arguments, and thus has $O(n)$ work and $O(\log n)$ span, where n is the length of s. Each call to `sum` inside the `map` is on a sequence of length n, and thus takes $O(n)$ work. This is mapped across n rows, yielding $O(n^2)$ work for the `map`. The final column is again of length n, so the final `sum` contributes $O(n)$, which is subsumed by the $O(n^2)$. However, the span is $O(\log n)$: the `map` doesn't contribute anything, and both the inner and outer `sum`s are on sequences of length n, and therefore are $O(\log n)$.

Tabulate The way of introducing a sequence is *tabulate*, which constructs a sequence from a function that gives you the element at each position, from 0 up to a specified bound:

```
tabulate f n ==> <v0 , ... , v_(n-1)>
   if f 0 ==> v0
      f 1 ==> v1
      ...
      f (n-1) ==> v_(n-1)
```

The time for `tabulate` is analogous to `map`.

Other operations We may encounter some other operations on sequences:

```
  val toString : ('a -> string) -> 'a seq -> string
  val flatten : 'a seq seq -> 'a seq

  (* repeat n x = <x,x,...x> with length n
     repeat n x has O(n) work and O(1) span
  *)
  val repeat  : int -> 'a -> 'a seq

  (* truncates longer if not the same length

     zip(s1,s2) has O(min(length s1,length s2)) work
                    O(1) span
   *)
  val zip     : ('a seq * 'b seq) -> ('a * 'b) seq

  (* split k <x0,...xk-1,xk...xn> == (<x0,...xk-1>, <xk,...xn>)
     (so the left result has length k)
     if the sequence has at least k elements
     or raises Range otherwise

     on a sequence s of length n, split k s has
     O(n) work
     O(1) span
  *)
  val split   : int -> 'a seq -> 'a seq * 'a seq

  (* take k <x0,...xk-1,xk...xn> == <x0,...xk-1>
     drop k <x0,...xk-1,xk...xn> == <xk,...xn>
     if the sequence has at least k elements
     or raise Range otherwise

     take i s and drop i s have
     O(i) work
     O(1) span
  *)
  val take    : int -> 'a seq -> 'a seq
  val drop    : int -> 'a seq -> 'a seq

  (* O(1) work and span *)
  val empty : unit -> 'a seq

  (* cons x xs has O(length xs) work and O(1) span *)
  val cons  : 'a -> 'a seq -> 'a seq

  (* O(1) work and span *)
  val singleton : 'a -> 'a seq

  (* append s1 s2 has O(length s1 + length s2) work and O(1) span *)
  val append    : 'a seq -> 'a seq -> 'a seq
```

These can all be implemented using the operations described above, though it is sometimes useful to implement some of them in terms of the underlying implementation of sequences to achieve better time bounds.

Tabulate Examples How do you implement `cons` with tabulate? It's the sequence of length one more than `xs`, whose first element is `x`, and whose next elements are the elements of `xs` shifted by one:

```
fun cons (x : 'a) (xs : 'a Seq.seq) : 'a Seq.seq =
   Seq.tabulate (fn 0 => x
                  | i => Seq.nth (i - 1) xs)
                (Seq.length xs + 1)
```

Note that with `nth` and `tabulate` you can write very index-y array-like code. Use this sparingly: it's hard to read! E.g. never write

```
   Seq.tabulate (fn i => ... nth i s ...)
                (Seq.length s)
```

if the function doesn't otherwise mention `i`: you're reimplementing `map` in a hard-to-read way!

```
   Seq.map (fn x => ... x ...) s
```

Next time, we will see how to use sequences to perform an *n-body simulation*: Given the mass, position, and velocity of n celestial bodies, simulate their movement over time due to gravitational forces.

2 N-Body

We can use sequences to perform an *n-body simulation*: Given the mass, position, and velocity of n celestial bodies, simulate their movement over time due to gravitational forces. n-body simulations are used by astrophysicists—e.g. they provide evidence that there is a black hole in the center of the Milky Way. Similar ideas apply to other forces; e.g. Coulomb's law has a similar mathematical form as the law of gravitation, and is used in simulating protein folding.

Here, we consider the n-body problem for 2 dimensions. The key step in the simulation is to compute the acceleration on each body due to all of the others, given their masses and positions. This requires computing a quadratic number of interactions. However, it is highly parallelizable, as the acceleration of each body can be computed independently of the accelerations of all of the others, and, for a given body, the component of the acceleration on that body due to each other can be computed in parallel.

This is specified as follows: Let a_i be the acceleration of body i. From Newton's second law, $F_i = m_i a_i$, we get that $a_i = F_i/m_i$. The force due to gravity is the sum of the forces of each other body:

$$F_i = \sum_{j \neq i} F_{ij}$$

By Newton's law of universal gravitation,

$$F_{ij} = \frac{G m_i m_j}{r^2}$$

where r_{ij} is the distance from i to j and G is the gravitational constant ($\approx 6.67x10^{-11} N(m/kg)^2$). Plugging in to the second law, the denominator distributes inwards, and cancels the m_i in each summand[1]

$$\begin{aligned} a_i &= \textstyle\sum_{j \neq i} a_{ij} \\ a_{ij} &= \tfrac{G m_j}{r_{ij}^2} \end{aligned}$$

This gives the magnitude of the acceleration. The direction of the force on m_1 due to m_2 is the direction from m_1 to m_2. Thus, as vectors:

$$\begin{aligned} \mathbf{a_i} &= \textstyle\sum_{j \neq i} \mathbf{a_{ij}} \\ \mathbf{a_{ij}} &= \hat{\mathbf{d_{ij}}} \tfrac{G m_j}{|d_{ij}|^2} \end{aligned}$$

where $\mathbf{d_{ij}}$ is the vector from m_i to m_j, $|d_{ij}|$ is its magnitude, and $\hat{\mathbf{d_{ij}}}$ is the unit vector in that direction ($\mathbf{d_{ij}}/|\mathbf{d_{ij}}|$).

Our simulation won't work for collisions (there are other forces than gravity involved), so we can stipulate that no collisions are allowed. Thus, we can assume that overlapping bodies are identical and define:

$$\begin{aligned} \mathbf{a_i} &= \textstyle\sum_{j} \mathbf{a_{ij}} \\ \mathbf{a_{ij}} &= \hat{\mathbf{d_{ij}}} \tfrac{G m_j}{|d_{ij}|^2} \text{if } i \text{ and } j \text{ don't overlap, or 0 otherwise} \end{aligned}$$

[1] This observation is due to Galileo: there is a (probably apocryphal) story about him dropping balls of different masses from the Leaning Tower of Pisa to demonstrate that the acceleration due to gravity on an object is independent of that object's mass—contradicting Aristotle, who though heavier objects fall faster.

2.1 Points and vectors

First, we represent points in the plane by pairs of real numbers. For each point in the plane, there is a vector space whose origin is that point. We represent vectors a pair of real numbers, which is the location of the head of the vector relative to the origin—but the origin is not specified in the code. The reason for this is that we will keep the velocity and acceleration vectors of each body in the vector space whose origin is the current location of that planet. E.g. the velocity of Earth will have its tail wherever Earth is. So the tail of the vector is stored externally, as the location of the body.

```
type point = real * real
type vec = real * real
```

We need the following operations on points and vectors:

Given two points, we can compute the vector from the first to the second:

```
infixr 3 -->
(* X --> Y is the vector from X to Y.
   computed by Y - X componentwise
   *)
fun ((x1,y1) : point) --> ((x2,y2) : point) : vec = (x2 - x1 , y2 - y1)
```

The result of `X --> Y` is in the vector space located at `X`.

We can also check whether two points have collided:

```
fun collided ((x1,y1) : point , (x2,y2) : point) : bool =
    Real.==(x1,x2) andalso Real.==(y1,y2)
```

To avoid floating point issues, it might be better to compare up to some small ϵ.

It does not make sense to add points, but it does make sense to displace a point by a vector in the vector space at that point, yielding the point at which the head of the vector is located.

```
(* assumes the vector is in the vector space of the point *)
fun displace ((x,y) : point , (x',y') : vec) : point = (x + x' , y + y')
```

Vectors have zero, addition, and scalar multiplication:

```
val zero : vec = (0.0 , 0.0)
infixr 3 ++
infixr 4 **
fun ((x1,y1) : vec) ++ ((x2,y2) : vec) : vec = (x1 + x2 , y1 + y2)
fun ((x,y) : vec) ** c : vec = (c * x , c * y)
```

We can compute the magnitude of a vector, as well as the unit vector in a direction:

```
fun mag ((x,y) : vec) : real = Math.sqrt (x * x + y * y)
fun unitVec (v : vec) : vec = v ** (1.0 / mag v)
```

In summary, we have the following operations on vectors:

```
type point = real * real
type vec = real * real
infixr 3 -->
val --> = fn : point * point -> vec
val collided : point * point -> bool
val displace : point * vec -> point
infixr 3 ++
infixr 4 **
val ++ : vec * vec -> vec
val ** : vec * real -> vec
val zero : vec
val mag : vec -> real
val unitVec : vec -> vec
```

Next week, we will talk about *modules*: given a problem domain (like points and vectors), you make a module collecting the types and operations that are useful for that domain. What we have just given is a start on a module for points and vectors.

2.2 N-body

Throughout, we use SI units: mass is in kg, length is in meters, and time is in seconds—so force is Newtons.

A body is represented by its mass, position (a point), and velocity (a vector).

```
(* mass, position, velocity *)
type body = real * point * vec
```

Accelerations The acceleration on m_i due to m_j, which we were calling $\mathbf{a_{ij}}$ above, is a simple transcription of the above equation:

```
val G = 6.67428E~11 (* N (m/kg)^2 *)

(* acceleration on body 1 due to body 2;
   note: this is independent of the velocity of the body *)
fun accOn ((_ , pos1 , _) : body , (m2 , pos2 , _) : body) : vec =
    case collided(pos1 , pos2) of
        true => zero
      | false =>
            let
                val d12 = (pos1 --> pos2)
            in
                unitVec d12 ** (G * m2 / (square (mag d12)))
            end
```

Next we compute the total acceleration of each body, i.e. the sequence $\langle \mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_n} \rangle$:

```
fun accelerations (bodies : body Seq.seq) : vec Seq.seq =
    Seq.map (fn b1 => sum bodies (fn b2 => accOn (b1 , b2))) bodies
```

`sum` is code implementing $\sum$: given a sequence of things, and a way to turn each thing into a vector, we can take the sum over the sequence:

```
fun sum (s : 'a Seq.seq) (f : 'a -> vec) : vec =
    Seq.mapreduce f zero op++ s
```

This implements the math

$$\sum_{x \in s} f x$$

Higher-order functions let us abstract the pattern $\sum$, rather than having to write it out individually each time.

Update Given an acceleration, we can update a body using basic mechanics:

$$\begin{aligned} p' &= p + vt + \tfrac{1}{2}at^2 \\ v' &= v + at \end{aligned}$$

```
(* a is the acceleration *)
fun stepBody ((m , p , v) : body , a : vec , t : real) : body =
    (m ,
     displace (p , v ** t ++ a ** (0.5 * t * t)),
     v ++ a ** t)
```

Finally, to update the whole universe, we first compute the acceleration of each body, and then update each body using its acceleration:

```
(* t is the timestep *)
fun step (bodies : body Seq.seq , t : real) =
    Seq.map (fn (b,a) => stepBody (b,a,t))
            (Seq.zip (bodies,
                      accelerations bodies))
```

Work/Span The work of accelerations is $O(n^2)$: the essence of it is a `mapreduce` (to compute and sum the acceleration on a body due to each other) inside of a `map` (do it for each body), and the arguments to the inner mapreduce are constant time. The span is $O(\log n)$: the `maps` are all constant span, and the only chain of dependencies is the `reduce` part of the `mapreduce` in `sum` used to add up up the components of the acceleration.

Thus, this algorithm is highly parallelizable. Where did this parallelism come from? First, from expressing the algorithm as maps and reduces on sequences of bodies, exploiting aggregates. Second, from expressing the inner computations as mathematical calculations (in this case, calculations on real numbers), which are all independent of one another. We hope you can see the close correspondence between the math in question and the code, which is important both for parallelism, and for elegance/readability of the code.

Barnes-Hut In homework, we're going to have you implement a faster but approximate way of computing accelerations. The idea is to summarize the action of far-away bodies by the action of a single point at their center of mass: don't compute the interaction between the Earth and each star in the Andromeda Galaxy; just summarize it. This results in a less precise, but faster simulation. The idea is to divide space up into regions, and compute summary information (position of center of mass, total mass) of each region. Then, when you are calculating the acceleration on a body, for regions that are "far enough" away (where "far enough" is a parameter), you use the summary information. This is called the Barnes-Hut algorithm, and its work is $O(n \log n)$ (assuming a good choice of "far enough") instead of quadratic.