15-150 Lecture 7: Parallel Sorting

Lecture by Dan Licata

February 7, 2012

Today's main points:

- Writing recurrence relations for span
- Lists are a bad data structure for parallelism
- Programming with trees

1 Span

In sequential evaluation, to evaluate a pair `(e1,e2)` you evaluate `e1` and *then* evaluate `e2`. Work is the number steps it takes to sequentiallly evaluate a program. In particular, this means that the work to evaluate `(e1,e2)` is the number steps to evaluate `e1` *plus* the number of steps to evaluate `e2`.

Parallel evaluation has all of the same rules as sequential evaluation, except to evaluate a pair `(e1,e2)` you evaluate `e1` and `e2` in parallel. E.g. `(3+1,4+5) |=> (4,9)` in one parallel step. (Caveat: SML is not actually implemented using parallel evaluation for pairs; to get parallelism, you need to use a special language construct (sequences), that we will talk about later in the semester. But all of the examples that we do using pairs can be rephrased to use sequences later.) Span is the number steps it takes to evaluate a program according to parallel evaluation. In particular, this means that the span to evaluate `(e1,e2)` is the *max* of the number of steps to evaluate `e1` and the number of steps to evaluate `e2`. The span is however long it takes whichever one takes longer to finish.

Recall the mergesort code from last lecture:

```
fun split (l : int list) : int list * int list =
    case l of
        [] => ([] , [])
      | [ x ] => ([ x ] , [])
      | x :: y :: xs => let val (pile1 , pile2) = split xs
                        in (x :: pile1 , y :: pile2)
                        end

fun merge (l1 : int list , l2 : int list) : int list =
    case (l1 , l2) of
        ([] , l2) => l2
      | (l1 , []) => l1
```

```
      | (x :: xs , y :: ys) =>
            (case x < y of
                 true => x :: (merge (xs , l2))
               | false => y :: (merge (l1 , ys)))

fun mergesort (l : int list) : int list =
    case l of
        [] => []
      | [x] => [x]
      | _ => let val (pile1,pile2) = split l
             in
                 merge (mergesort pile1, mergesort pile2)
             end
```

The work was $O(n \log n)$.

What is the span? You might think that mergesort is good for parallelism, because of the tree-shaped decomposition of problems:

```
           sort [3,1,4,2]
   sort [3,1]           sort [4,2]
sort [3]  sort [1]   sort [4]   sort [2]
```

The depth of the tree is $O(\log n)$, where n is the length of the list. So maybe mergesort has logarithmic span?

Unfortunately, no! The very first thing it does it to use `split` to deal the list out into two piles. There is no parallelism here, since it deals the elements out one by one, so we have to wait at least $O(n)$ timesteps right at the start, even if we have as much computational power as we need. This is bad.

We just argued that the span is at least linear. Let's write a recurrence to see what it actually is. The recurrences for `split` and `merge` are the same as they would be for the work, because there are no pairs to evaluate in parallel:

$$\begin{aligned} S_{split}(n) &= k_1 + S_{split}(n-2) \\ S_{merge}(n) &= k_2 + S_{merge}(n-1) \end{aligned}$$

Thus, both are $O(n)$.

However, for mergesort, we get

$$\begin{aligned} S_{mergesort}(n) &= k_3 + S_{merge}(n) + S_{split}(n) + max(S_{mergesort}(n/2), S_{mergesort}(n/2)) \\ &\leq k' \cdot n + \cdot S_{mergesort}(n/2) \end{aligned}$$

Because the two recursive calls are in a pair, they can be calculated in parallel, and thus we only count one of them towards the span.

If we factor out the constant and expand out this recurrence, we get

$$\begin{aligned} S_{mergesort}(n) &\leq k'(n + n/2 + n/4 + n/8 + n/16 + \ldots) \\ &= k'(n + n \cdot (\Sigma_{i=1}^{\log n}(1/(2^i)))) \end{aligned}$$

The latter sum is always less than 1 (this is Zeno's paradox: if you always step halfway there, you never get there), so the whole thing is less than $2k'n$, and $S_{mergesort}$ is therefore $O(n)$.

This is less than ideal. For the moment, ignore the constant factors; a similar example can be chosen no matter what they are. Suppose you want to sort a billion numbers on 64 processors. $\log 10^9$ is about 30. So the total work to do is 30 billion. On 64 processors, this should take less than half a billion timesteps, if you divide the work perfectly among all 64 processors. However, this span says that the length of the longest critical path is still a billion, so you can't achieve this division! This gets worse as the number of processors gets larger.

The problem is that *lists are bad for parallelism.* The list data structure does not admit an efficient enough implementation of split and merge to exploit all the parallelism one might have.

2 Trees

We can solve this problem by switching to a different data structure: *the choice of data structure influences how much parallelism is available.* We will do a version of mergesort on trees that still has $O(n \log n)$ work (so it's *work-efficient*: the work is the same as the sequential algorithm you start with) and has $O((\log n)^3)$ span.

2.1 Lists as a Datatype

Lists are built in to ML, and we have said that

> A list of integers (`int list`) is either
>
> > `[]`, or
> >
> > `x :: xs` where `x : int` and `xs : int list`.
>
> And that's it!

However, aside from some syntactic issues, lists could be defined by the following datatype definition:

```
datatype intlist =
    []
  | :: of int * intlist
```

Caveats: you can't actually use `[]` as a constructor name; you can say that constructors should be used in infix notation (to write `x :: xs`), but this doesn't; the bracket notation `[1,2,3]` is special to lists. But other than that, lists are just a datatype.

This generates *constructors*, `[]` and `::`, which can be used both to construct lists and to case-analyze them, as we have been doing.

2.2 Tree Datatype

Trees are defined as follows:

> A `tree` is either
>
> > `Empty`, or
> >
> > `Node(l,x,r)` where `x : int` and `l : tree` and `r : tree`.
>
> And that's it!

Trees are constructed by applying constructors:

```
val example = Node (Empty , 1 , Node (Empty , 3 , Empty))
```

However, trees are not built-in, so we have to define them. We do this using a datatype declaration:

```
datatype tree =
    Empty
  | Node of (tree * int * tree)
```

This means the same thing as the English above, but is in a form SML can understand. It defines a new type `tree` with two constructors, `Empty` and `Node`. Values can be constructed as above, and the operation is, as usual, case-analysis and recursion:

```
(* Purpose: compute the number of elements in the tree *)
fun size (t : tree) : int =
    case t of
        Empty => 0
      | Node (l, x, r) => 1 + size l + size r
val 2 = size example
```

This illustrates *structural recursion on trees*: Because there are two recursive occurrences of `tree` as arguments to `Node`, you get two recursive calls, one for each.

3 Sorted Trees

When is a tree sorted?

- `Empty` is sorted.
- `Node(l,x,r)` is sorted iff
  - `l` is sorted, and
  - `r` is sorted, and
  - everything in `l` is `<= x`, and
  - `x <` everything in `r`.
- `e` is sorted if `e` $\cong$ `e'` and `e'` is sorted.

The first two clauses say when values are sorted; the third says that sortedness respects equivalence.

For example,

```
Node(Node(Node(Empty,1,Empty),
          2,
          Node(Empty,3,Empty)),
     4,
     Node(Empty,5,Empty))
```

is sorted using the first two clauses. We'll use the third to state a theorem like "for all trees `t`, `sort t` is sorted" and then prove it by calculating.

4 Mergesort on Trees

4.1 Code

Let's be bold and follow the template for structural recursion:

```
fun mergesort (t : tree) : tree =
    case t of
        Empty => Empty
      | Node (l , x , r) => ... mergesort l ... mergesort r ...
```

Assuming we have sorted `l` and `r`, what do we need to finish off the case? We need to *merge* together the two sorted results and the tree containing just `x`. So we push one helper function on the to-do list:

```
(* Purpose: combine two sorted trees into a third, containing exactly
            the elements of both *)
fun merge (t1 : tree , t2 : tree) : tree = ...
```

and finish off mergesort by calling it:

```
fun mergesort (t : tree) : tree =
    case t of
        Empty => Empty
      | Node (l , x , r) =>
          merge(merge (mergesort l , mergesort r),
                Node(Empty,x,Empty))
```

We use `Node(Empty,x,Empty)` to make a one-element tree, and use `merge` twice to put these three trees together.

Why is this better than mergesort for lists? First, the split into subproblems is *constant time*—the splitting is given by the data structure itself! Second, we can merge two trees in sublinear span, which gets a sublinear span overall. This is tricky, but doable; we'll show how today.

Merging Here's the idea with merging: say we need to merge

```
Node (Node(Empty,1,Empty), 3 , Node(Empty,5,Empty))
and
Node (Node(Empty,2,Empty), 4 , Node(Empty,6,Empty))
```

We are, somewhat arbitrarily, going to choose to be guided by the first tree, and will stipulate that the root of the first tree will be the overall root. So the question is, how do we need to fill in this:

```
Node (merge(?,?) , 3 , merge(?,?))
```

Clearly, the left subtree of `3` needs to go to the left, and the right to the right, for the result to be sorted and contain all the appropriate elements.

```
Node (merge(Node(Empty,1,Empty),?) , 3 , merge(Node(Empty,5,Empty),?))
```

For similar reasons, we need to put everything in the second tree that is less than `3` to the left, and everything greater to the right.

```
Node (merge(Node(Empty,1,Empty),Node(Empty,2,Empty)),
      3,
      merge(Node(Empty,5,Empty),Node (Empty, 4 , Node(Empty,6,Empty))))
```

So let's make up another helper function:

```
(* Purpose: assuming t is sorted, split t along the bound,
            returning (l,r) where
            l contains the elts of t that are <= bound
            r contains the elts of t that are > bound
            and both l and r are sorted *)
fun splitAt (t : tree , bound : int) : tree * tree =
```

Using this, it's simple to write merge:

```
fun merge (t1 : tree , t2 : tree) : tree =
 case t1 of
     Empty => t2
   | Node (l1 , x , r1) =>
      let val (l2 , r2) = splitAt (t2 , x)
      in
          Node (merge (l1 , l2) ,
                x,
                merge (r1 , r2))
      end
```

Split Here's the code for `split`:

```
fun splitAt (t : tree , bound : int) : tree * tree =
 case t of
     Empty => (Empty , Empty)
   | Node (l , x , r) =>
      (case bound < x of
           true => let val (ll , lr) = splitAt (l , bound)
                   in (ll , Node (lr , x , r))
                   end
         | false => let val (rl , rr) = splitAt (r , bound)
                    in (Node (l , x , rl) , rr)
                    end)
```

Let's look at the case where `bound < x`. By induction `splitAt(l,bound)` divides up `l` so that everything less than `bound` is in `ll` and evertying greater is in `lr`. We know that `x` and `r` have to go on the right side, because `x > bound`, everything in `r` is greater than `x`, and therefore also greater than the bound. The other case is symmetric.

4.2 Correctness

How would you prove that `mergesort` returns a sorted tree? *Structural induction on trees!*

Theorem 1. *For all trees* `t`*,* `mergesort t` *is sorted.*

Here's the template:

Proof. **Case for Empty**
To show: `mergesort Empty` is sorted.
Proof: `mergesort Empty == Empty` in 2 steps, and `Empty` is sorted by defintion.
Case for `Node(l,x,r)`
Inductive hypotheses: (1) `mergesort l` is sorted.
(2) `mergesort r` is sorted.

To show: `mergesort (Node (l,x,r))` is sorted.
Proof:

```
   mergesort (Node (l,x,r))
== merge (merge (mergesort l, mergesort r), Node(Empty,x,Empty))   [step x 2]
```

By the IH, `mergesort l` and `mergesort r` are sorted. By the spec for `merge`, which says that `merge` takes two sorted trees to a third, `merge(mergesort l, mergesort r)` is sorted. Applying this spec again, `merge (merge (mergesort l, mergesort r), Node(Empty,x,Empty))`, so because sortedness respects equivalence, `mergesort (Node (l,x,r))` is sorted as well. □

It's worth pointing out a subtlety in the proof: The easiest thing to prove about `merge` is

Lemma 1. *For all values* `l:tree` *and* `r:tree`*, if* `l` *is sorted and* `r` *is sorted then* `merge(l,r)` *is sorted.*

When we state the theorem for *values*, then we can do a proof by induction, because the values of type `tree` are indutively defined.

However, in the proof, we need to appeal to the lemma on non-values, like `mergesort l` and `mergesort r`. To do this, we can lift the lemma to *valuables*:

Corollary 1. *For all valuable expressions* `e1:tree` *and* `e2:tree`*, if* `e1` *is sorted and* `e2` *is sorted then* `merge(e1,e2)` *is sorted.*

Proof. By definition of valuability `e1` $\cong$ `v1` and `e2` $\cong$ `2` for some values `v1` and `v2`. Because sorted respects equivalence, `v1` and `v2` are sorted. By the lemma above, this means that `merge(v1,v2)` is sorted. Again because sorted respects equivalence, `merge(e1,e2)` is sorted. □

In the proof, we then need to know that `mergesort e1` and `mergesort e2` and `merge(mergesort e1,mergesort e2)` are valuable. But it is simple to prove that all of `splitAt`, `merge`, and `mergesort` are total, because they are structurally recursive.

4.3 Analysis

The overall work of mergesort is $O(n \log n)$, and the span is $O((\log n)^3)$. Let's look at the span analysis.

First, let d be the depth of the tree. Then

$$S_{splitAt}(d) \quad = \quad k + S_{splitAt}(d-1)$$

because peeling off the `Node` constructor decreases the depth by 1. Thus, $S_{splitAt}(d)$ is $O(d)$.

Let d_1 and d_2 be the depths of `t1` and `t2`. Then

$$S_{merge}(d_1, d_2) \quad = \quad k + S_{splitAt}(d_2) + max(S_{merge}(d_1 - 1, d_{21}), S_{merge}(d_1 - 1, d_{22}))$$

where d_{21} and d_{22} are the depths of the result of the split. These are no deeper than the original tree, so we can overapproximate as

$$\begin{aligned} S_{merge}(d_1, d_2) &\leq k + S_{splitAt}(d_2) + max(S_{merge}(d_1 - 1, d_2), S_{merge}(d_1 - 1, d_2)) \\ &\leq k'd_2 + S_{merge}(d_1 - 1, d_2) \end{aligned}$$

Expanding this out, you can see that we do $k'd_2$ work d_1 times, so $S_{merge}(d_1, d_2)$ is $O(d_1 \cdot d_2)$.

Now for mergesort. Let n be the *size* of the tree, and assume it is balanced, so the depth is about $\log n$.

$$\begin{aligned} S_{mergesort}(n) &= k + max(S_{mergesort}(n/2), S_{mergesort}(n/2)) + S_{merge}(\log n, \log n) + S_{merge}(2\log n, 1) \\ &\leq k + S_{mergesort}(n/2) + k_1(\log n)^2 + k_2 \log n \\ &\leq S_{mergesort}(n/2) + k_3(\log n)^2 \end{aligned}$$

The second call to `merge` is on the output of the first, so we need to know how `merge` changes the depth of the tree. Fortunately, we can prove that

$$\texttt{depth (merge(l,r))} \leq \texttt{depth l} + \texttt{depth r}$$

Overall the recurrence says that we do $O((\log n)^2)$ steps (the divisions inside the log don't help you, because it's just subtracting off a constant) $\log n$ times, so the overall span is $O((\log n)^3)$. Thus (ignoring constants), when we try to sort a billion elements, the length of the longest critical path is about 27000 operations, so we can exploit over a million processors!

Or it would be, if there wasn't a bug in this analysis! When instantiated $S_{merge}(\log n, \log n)$ and $S_{merge}(\log n, 1)$, we were relying on the fact that the trees we passed to `merge` were balanced. However, this is not necessarily the case, because we call these functions on the *output* of mergesort. Moreover, mergesort doesn't necessarily produce a balanced tree. We can fix this by rebalancing each time through:

```
fun mergesort (t : tree) : tree =
 case t of
    Empty => Empty
  | Node (l , x , r) =>
        rebalance(merge(merge (mergesort l , mergesort r),
                        Node(Empty,x,Empty)))
```

You'll look at the code for rebalance in the homework; it doesn't change the overall work or span.