Parallel Mergesort
Justin Raymond
December 14, 2015

1 Description

1.1 Specification

Parallel mergesort is an algorithm for sorting a list in parallel. We use a tree because splitting a tree is a constant time operation. Splitting a list has complexity linear in the size of the list, so the divide step of mergesort takes is linear in the size of the list, increasing the parallel overhead.

We use binary trees with a sized cached for each node.

```
type 'a tree = | E | N of 'a * int * 'a tree * 'a tree;;
```

We define a *sorted tree* as follows.

1. `E` is sorted.
2. `N(x,s,l,r)` is sorted if `l` is sorted, `r` is sorted and `max(l) <= x < min(r)`.

Our analysis requires that the trees passed as arguments to some functions be `balanced`. We use the following notion of balance.

1. `E` is balanced.
2. `N(x,s,l,r)` is balanced if `l` is balanced, `r` is balanced, and `abs(size l - size r) < 2`.

The function `mergesort` applied to a tree `t` will return a sorted, balanced tree `t'`.

1.2 Implementation and Analysis

1.2.1 `size`

We use the function `size` to determine the size of a tree. An empty tree has a size of 0. The size of a node is cached at the node itself.

```
1
2  let size t =
3    match t with
4    | E -> 0
5    | N(_, s, _, _) -> s;;
```

The work and span of `size` is obviously constant.

1.2.2 `take_and_drop`

The function `take_and_drop` divides a tree into two trees, preserving the order of the elements. The left and right trees either have the same size, or the right tree has size one greater than the left subtree.

```
1  let rec take_and_drop n t =
2    match (n, t) with
3    | (0, _) -> (E, t)
4    | (_, E) -> (E, E)
5    | (_, N(x, _, l, r)) ->
6        if n <= size l
7        then
8          let (t, d) = take_and_drop n l in
9          (t, N(x, 1 + size d + size r, d, r))
10       else
11         let (t, d) = take_and_drop (n - 1 - size l) r in
12         (N(x, 1 + size l + size t, l, t), d);;
```

We analyze the work and span of `take_and_drop` in terms of the depth of the argument tree. We either are at the base case or make one recursive call. At each recursive call we do some constant amount of work. So

$$W_{tad}(d) = \begin{cases} k_1 & \text{if } d = 0 \\ k + W_{tad}(d-1) & \text{otherwise} \end{cases}$$

The solution to this recurrence is $W_{tad}(d) \leq kd + k_1$. Similarly, the span is $S_{tad}(n, d) \leq kd + k_1$.

1.2.3 `del_min`

The function `del_min` deletes the smallest element from a sorted tree and returns the element along with the remaining tree. `del_min` requires the size of the input tree to be greater than zero.

```
1 let rec del_min t =
2   match t with
3   | E -> raise INVARIANT_FAILED
4   | N(x, _, E, r) -> (x, r)
5   | N(x, s, l, r) ->
6       let (m, l') = del_min l in
7       (m, N(x, s - 1, l', r))
```

The work depends on the length of the path to the leftmost child, which in the worst case is the depth of the tree.

$$W_{del}(d) = \begin{cases} k_1 & \text{if left child is } \texttt{E} \\ k + W_{del}(d-1) & \text{otherwise} \end{cases}$$

The solution to this recurrence is $W_{del}(d) \leq kd + k_1$. Similarly the span is $S_{del}(d) \leq kd + k_1$.

1.2.4 `rebalance`

The function `rebalance` takes tree and returns a balanced tree, preserving the order of the elements. Our rebalancing strategy is simple. First we divide a potentially unbalanced tree into two trees whose sizes differ by at most one. Then we retrieve the minimum of the larger tree to serve as the root of a new tree. Finally, we recursively repeat this operation on the left and right subtrees of the new tree. We use `take_and_drop` to divide the tree into two subtrees whose size differ by at most one. We delete the minimum element of the right subtree and use it to create a new root. Then we recursively rebalance the left and right subtrees.

```
1 let rec rebalance t =
2   match t with
3   | N(x, s, l, r) ->
4       let (taken, dropped) = take_and_drop (s / 2) t in
5       (match dropped with
```

```
6         | E -> taken
7         | _ ->
8           let (root, dropped') = del_min dropped in
9           N(root, s, rebalance taken, rebalance dropped'))
10    | _ -> t;;
```

The work of `rebalance` is

$$W_{reb}(n) = \begin{cases} k_1 & \text{if } n = 0 \\ k + W_{tad}(n) + W_{del}(n) + W_{reb}(n/2) + W_{reb}(n/2) & \text{otherwise} \end{cases}$$

For $n > 0$, $W_{reb}(n) \leq k+2n+2W_{reb}(n/2)$. We can use the master method to solve this recurrence. The recurrence is of the form $T(n) = aT(\frac{n}{b}) + f(n)$, where $a = 2, b = 2$ and $f(n) = 2n$. Since $f(n) = \Theta(n^{\log_2 2}) = \Theta(n)$, $W_{reb}(n) = \Theta(n \log n)$.

We can make the recursive calls to rebalance in parallel, so the span will be less than the work.

$$S_{reb}(n) = \begin{cases} k_1 & \text{if } n = 0 \\ k + S_{tad}(n) + S_{del}(n) + max(S_{reb}(n/2), S_{reb}(n/2)) & \text{otherwise} \end{cases}$$

For $n > 0$, $S_{reb}(n) \leq k + 2n + S_{reb}(n/2)$. Again we can use the master method, with $a = 1, b = 2$, and $f(n) = 2n$. The third case of the master method applies. With $\epsilon = 1$, $f(n) = \Omega(n^{\log_2 1+\epsilon}) = \Omega(n)$. We must also verify $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n. This holds for $c = 1$ and all $n > 0$. So $S_{reb}(n) = \Theta(n)$.

1.2.5 `split_at`

The function `split_at` is used to help merge two trees. `split_at x t` returns a pair of trees, the first contains all elements in `t` less than or equal to `x`, the second contains elements strictly greater than `x`. Observe that the output is not guaranteed to be balanced.

```
1  let rec split_at x t =
2    match t with
3    | E -> (E, E)
4    | N(y, _, l, r) when y <= x ->
5        let (ltx, gtx) = split_at x r in
```

```
6         (N(y, 1 + size ltx + size l, l, ltx), gtx)
7     | N(y, _, l, r) ->
8         let (ltx, gtx) = split_at x l in
9         (ltx, N(y, 1 + size gtx + size r, gtx, r));;
```

We will assume the input tree is balanced. The work is expressed by the recurrence

$$W_{split}(d) = \begin{cases} k_1 & \text{if } d = 0 \\ k + W_{split}(d-1) & \text{otherwise} \end{cases}$$

So $W_{split}(d) = kd + k_1$. Similarly the span is $S_{split}(d) = kd + k_1$, since there is no opportunity to run recursive calls in parallel.

1.2.6 `merge`

The function `merge` takes two sorted trees `t1` and `t2` and returns the sorted tree containing all elements of `t1` and `t2`. We split `t2` into two trees, let us call them `ltx` and `gtx`, using `x`, the value at the root node of `t1`. We then return a tree with `x` at the root, the result of recursively merging `ltx` and the left child of `t1` as the left child, and the result of recursively merging `gtx` and the right child of `t1` as the right child.

```
1 let rec merge t1 t2 =
2   match t1 with
3     | E -> t2
4     | N(x, _, l, r) ->
5         let (ltx, gtx) = split_at x t2 in
6         N(x, size t1 + size t2, merge l ltx, merge r gtx);;
```

We assume the arguments are balanced. The work is

$$W_{merge}(d_1, d_2) = \begin{cases} k_1 & \text{if } d = 0 \\ k + W_{split}(d_2) + W_{merge}(d_1 - 1, d_2) + W_{merge}(d_1 - 1, d_2) & \text{otherwise} \end{cases}$$

So `merge` does $kd_2 + k$ work at each step of recursion. Since we make two recursive calls at each level, and the are d_1 levels in the recursion tree, $W_{merge} = \mathcal{O}(d_2 2^{d_1})$. Notice that the order of the arguments to `merge` are important. We want to call `merge` with the shorter tree as the first argument, because it is the argument `merge` recurses on.

We can make the recursive calls in parallel, so the span is

$$S_{merge}(d_1, d_2) = \begin{cases} k_1 & \text{if } d_1 = 0 \\ k + S_{split}(d_2) + max(S_{merge}(d_1 - 1, d_2), W_{merge}(d_1 - 1, d_2)) & \text{otherwise} \end{cases}$$

For $d_1 > 0$, $S_{merge}(d_1, d_2) = k + d_2 + S_{merge}(d_1 - 1, d_2)$. So the $S_{merge}(d_1, d_2) = \mathcal{O}(d_1 * d_2)$.

1.2.7 `mergesort`

The function `mergesort` recursively sorts it's children, then merges the results and the value at its root. We do not need to do the split present in array-based mergesort, because the split is already present in the data structure itself. We balance the resulting tree because the running time of `merge` depends on its input being balanced.

```
1 let rec mergesort t =
2   match t with
3   | E -> E
4   | N(x, _, l, r) ->
5       rebalance (merge (N(x, 1, E, E)) (merge (mergesort l) (mergesort r)))
```

The work is

$$W_{ms}(n) = \begin{cases} k_1 & \text{if } n = 0 \\ k + W_{merge}(\log n, \log n) + W_{merge}(1, 2\log n) + W_{reb}(n) + 2W_{ms}(\frac{n}{2}) & \text{otherwise} \end{cases}$$

The result of the recursive calls will be balanced, so we will make the first call to `merge` with two trees of height $\log n$. In the worst case, the result will have height $2\log n$. So the second call to `merge` will be on a trees of height 1 and $2\log n$. Therefore for $n > 0$, $W_{ms}(n) = n\log n + 4\log n + n\log n + 2W_{ms}(\frac{n}{2})$. We cannot apply the master method since $f(n) = 6n\log n$ is not polynomially larger than n. Instead we will use the substitution method. We guess $W_{ms}(n) = kn^2\log n + k_1$. If $n = 0$, then $W_{ms}(n) = k_1$. If $n > 0$, then $W_{ms}(n) = 6n\log n + 2W_{ms}(\frac{n}{2})$. We use the induction hypothesis to get $W_{ms}(n) = 6n\log n + 2k(\frac{n}{2})^2\log\frac{n}{2} + 2k_1$. We can simplify to $W_{ms}(n) = 6n\log n + k(\frac{n^2}{2})(\log n - \log 2) + 2k_1$.

The recursive calls can be made in parallel, so the span is

$$S_{ms}(n) = \begin{cases} k_1 \text{ if } n = 0 \\ k + S_{merge}(\log n, \log n) + S_{merge}(1, 2\log n) + W_{reb}(n) + max(S_{ms}(\frac{n}{2}), S_{ms}(\frac{n}{2})) \end{cases}$$

For $n > 0$, $S_{ms}(n) = k + (\log^2 n) + 2 \log n + n + S_{ms}(\frac{n}{2})$. We see that we have $\log^2 n$ cost $\log n$ times, so $S_{ms} = \mathcal{O}(\log^3 n)$.

2 Code

```
1  type 'a tree = | E | N of 'a * int * 'a tree * 'a tree;;
2
3  let size t =
4    match t with
5    | E -> 0
6    | N(_, s, _, _) -> s;;
7
8  let rec del_min t =
9    match t with
10   | E -> raise INVARIANT_FAILED
11   | N(x, _, E, r) -> (x, r)
12   | N(x, s, l, r) ->
13       let (m, l') = del_min l in
14       (m, N(x, s - 1, l', r))
15
16 let rec split_at x t =
17   match t with
18   | E -> (E, E)
19   | N(y, _, l, r) when y <= x ->
20       let (ltx, gtx) = split_at x r in
21       (N(y, 1 + size ltx + size l, l, ltx), gtx)
22   | N(y, _, l, r) ->
23       let (ltx, gtx) = split_at x l in
24       (ltx, N(y, 1 + size gtx + size r, gtx, r));;
25
26 let rec merge t1 t2 =
27   match t1 with
28   | E -> t2
29   | N(x, _, l, r) ->
30       let (ltx, gtx) = split_at x t2 in
31       N(x, size t1 + size t2, merge l ltx, merge r gtx);;
32
```

```
33 let rec take_and_drop n t =
34   match (n, t) with
35   | (0, _) -> (E, t)
36   | (_, E) -> (E, E)
37   | (_, N(x, _, l, r)) ->
38       if n <= size l
39       then
40         let (t, d) = take_and_drop n l in
41         (t, N(x, 1 + size d + size r, d, r))
42       else
43         let (t, d) = take_and_drop (n - 1 - size l) r in
44         (N(x, 1 + size l + size t, l, t), d);;
45
46 let rec rebalance t =
47   match t with
48   | N(x, s, l, r) ->
49       let (taken, dropped) = take_and_drop (s / 2) t in
50       (match dropped with
51       | E -> taken
52       | _ ->
53         let (root, dropped') = del_min dropped in
54         N(root, s, rebalance taken, rebalance dropped'))
55   | _ -> t;;
56
57 let rec mergesort t =
58   match t with
59   | E -> E
60   | N(x, _, l, r) ->
61       rebalance (merge (N(x, 1, E, E)) (merge (mergesort l) (mergesort r))
```