Notes on the CpSc 418 Course Erlang Library

Draft version: no bug-bounties (yet) for minor spelling and grammar errors.

1 Distributed Lists

In Erlang, lists are the primary data structure. To write efficient parallel code, we need each process to store data locally as much a possible. This means we need to be able to take a large list, and distribute across the processes. The `workers` library provides functions `workers:update` `workers:retrieve` to do this. The function `misc:cut` in the `misc` module is helpful as well.

Let's say that `L101` is a long list. For this example, `L101` will be the list `lists:seq(1,101)`, i.e. `[1,2,ldots,101]` – that's not very "long", but it means I can write out the example without using hundreds of pages to print the list. Let `W8` be a worker-tree with eight processes. We can partition `L101` into eight smaller lists using `misc:cut`:

```
misc:cut(L101, W8) ->
  [ [1,2,3,4,5,6,7,8,9,10,11,12],                  % 12 elements
    [13,14,15,16,17,18,19,20,21,22,23,24],         % 12 elements
    [25,26,27,28,29,30,31,32,33,34,35,36],         % 12 elements
    [37,38,39,40,41,42,43,44,45,46,47,48,49],      % 13 elements
    [50,51,52,53,54,55,56,57,58,59,60,61,62],      % 13 elements
    [63,64,65,66,67,68,69,70,71,72,73,74,75],      % 13 elements
    [76,77,78,79,80,81,82,83,84,85,86,87,88],      % 13 elements
    [89,90,91,92,93,94,95,96,97,98,99,100,101]] % 13 elements
```

The first argument to `misc:cut` is the list to split up. The second argument tells `misc:cut` how many pieces to make. In a bit more detail,

- If the second argument to `misc:cut`(L, N) is an integer, then L is divided into N segments.
- If the second argument to `misc:cut`(L, W) is an list, then L is divided into `length(`W`)` segments.

Why have the second case? Because worker trees such as `W8` are represented as lists. The length of this list is the number of workers. Of course, I'd prefer that *you* didn't depend on knowing the internal data structures of the `workers` and `wtree` modules. In the not-too-likely event that I were to change the representation of worker trees, I would revise `misc:cut` to work with the new representation – I promise.

Now that we have divided `misc:cut` into pieces, we need to send these pieces to the workers. The function `workers:update` does just this. The general form is

```
workersupdate(W, Key, ListOfValues)
```

where `length(ListOfValues)` matches the number of workers in `W`. For example,

```
workersupdate(W8, my_list, misc:cut(L101, W8))
```

will divide `misc:cut`(L101) into eight pieces and send each process of `W8` a separate piece. The workers in the tree are ordered from left-to-right, and the values of `ListOfValues` are distributed in that order. In other words, the leftmost worker process receives the first value of `ListOfValues` and so on.

What is the `Key` all about? The workers need a way to access these values. For reduce, scan, and other operations, you will provide functions for these worker processes to access. How do they get the list you just sent? You will note that the *Leaf* functions for `wtree:reduce` or `wtree:scan` have one parameter, `ProcState`. `ProcState` is an association list that pairs keys with values. The worker processes access values in `ProcState` using the functions `wtree:get` and `wtree:put` (equivalently, `workers:get` and `workers:put`. In particular:

`wtree:get(ProcState, Key)` returns the value in ProcState associated with `Key`. If there is no value associated with `Key`, then the atom `undefined` is returned.

`wtree:get(ProcState, Key, Default)` returns the value in ProcState associated with `Key`. If there is no value associated with `Key`, then the value of the parameter `Default` is returned.

`wtree:put(ProcState, Key, Value)` associate `Key` with `Value` in `ProcState`. Wait a second – **Erlang is functional!** Right. `wtree:put` creates a new association list, let's call it `ProcState2`. `ProcState2` has all of the `{K, V}` mappings that were in `ProcState` plus the mappping `{Key, Value}`. If the mappping `{Key, OldValue}` was already present in `ProcState`, then `ProcState2` has all of the mappings of `ProcState` except for `{Key, OldValue}` and it has the mapping `{Key, Value}`. In other words, the mapping for `{Key, OldValue}` in `ProcState` is replaced with `{Key, Value}` in `ProcState2`.

2 Reduce and Scan

The functions `wtree:reduce` and `wtree:scan` make use of `ProcState` and that typically involves distributed lists as described above. First, let's look at `wtree:reduce(W, Leaf, Combine, Root)` where `W` is a worker-tree produced by `wtree:create`.

`Leaf(ProcState) -> Value`: `ProcState` is an association of keys to values for this process as described above. This is how the process finds the data that it is to work on. `Value` is the value that `Leaf` computes based on `ProcState` and returns for the `Combine` tree. We typically use an Erlang `fun()` expression to capture information about what key to use to access `ProcState` and any other details that `Leaf` may need. I should put an example here, but I'm running out of time.

`Combine(Left, Right) -> Value`: `Left` and `Right` are the values from the left and right subtees of this node. Combine them into a single value. Note that `Left`, `Right`, and `Value` should all be of the same "type". For example, if `Left` and `Right` are numbers and `Value` is a tuple, you're probably doing something wrong, and the call to `Combine` in the next level of the tree will probably crash. Note that `Combine` does not take `ProcState` as a parameter. Anything you need to know about `ProcState` should be included in the values returned by `Leaf` and `Combine`. However, if you're returning the entire segment of a distributed list for that leaf-node or subtree, you probably aren't doing the reduce correctly.

`Root(Value0) -> Value1`: again, no `ProcState`.

I'll continue this document with a description of `wtree:scan` and `wtree:rlist`. The key points are that the `Leaf2` function for `wtree:scan` should return an updated `ProcState`. The function `wtree:rlist` produces a random list that is distributed across the workers of a worker tree.