# Generalize Reduce and Scan

Mark Greenstreet

CpSc 418 – Jan. 17, 2018

Outline:

- Reduce in Erlang
- Scan in Erlang

# Objectives

- Understand relationship between reduce and scan
  - Both are tree walks.
  - The initial combination of values from leaves is identical.
  - Reduce propagates the grand total down the tree.
  - Scan propagates the total "everything to the left" down the tree.
- Generalized Reduce and Scan
  - Understand the role of the *Leaf*, `Combine`, and `Root` functions.
  - Understand the use use of higher-order functions to implement reduce and scan.
- The CS418 class library
  - Able to create a tree of processes.
  - Able to distribute data and tasks to those processes.
  - Able to use the `reduce` and `scan` functions from the library.
  - Know where to find more information.

# Reduce in Erlang

- Build a tree.
- Each process creates a lists of random digits.
- The processes meet at a barrier so we can measure the time to count the 3s.
- Each process counts its threes.
- The processes use reduce to compute the grand total.
- Each process reports the grand total and its own tally.
- The root process reports the time for the local tallies and the reduce.
- Get the code at

http://www.ugrad.cs.ubc.ca/~cs418/2017-2/lecture/01-16/code/reduce.erl

# The Reduce Pattern

- It's a parallel version of *fold*, e.g. `lists:foldl`.
- Reduce is described by three functions:

  *Leaf()*: What to do at the leaves, e.g.

  ```
  fun() -> count3s(Data) end
  ```

  *Combine()*: What to do at the root, e.g.

  ```
  fun(Left, Right) -> Left+Right end
  ```

  *Root()*: What to do with the final result. For count 3s, this is just the identity function.

# The `wtree` module

- Part of the course Erlang library.
- Operations on worker trees"

  `wtree:create(NProcs) -> [pid()].`

  Create a list of `NProcs` processes, organized as a tree.

  `wtree:broadcast(W, Task, Arg) -> ok.`

  Execute the function `Task` on each process in `W`. Note: `W` means "worker pool".

  `wtree:reduce(P, Leaf, Combine, Root) -> term().`

  A generalized reduce.

  `wtree:reduce(P, Leaf, Combine) -> term().`

  A generalized reduce where `Root` defaults to the identity function.

# Store Locally

- Communication is expensive – each process should store its own data whenever possible.
- How do we store data in a functional language?
  - Our processes are implemented as Erlang functions that receive messages, process the message, and make a tail-call to be ready to receive the next message.
  - We add a parameter to these functions, `ProcState`, that is a mapping from *Keys* to *Values*.
- What this means when we write code:

  Functions such as *Leaf* for `wtree:reduce` or *Task* for `wtree:broadcast` have a parameter for `ProcState`.

  `workers:put(ProcState, Key, Value) -> NewProcState.`

  Create a new version of `ProcState` that associates `Value` with `Key`.

  `workers:get(ProcState, Key, Default) -> Value.`

  Return the value associated with `Key` in `ProcState`. If no such value is found, `Default` is returned. Note: `Default` can be a function in which case it is called to determine a default value – see the documentation.

  `workers:get(ProcState, Key) -> workers:get(ProcState,`

# Count3s using `wtree`

```
count3s_par(N, P) ->
  W = wtree:create(P),
  wtree:rlist(W, N, 10, 'Data'),
  wtree:reduce(W,
    fun(ProcState) -> % Leaf
      count3s(workers:get(ProcState, data))
    end,
    fun(Left, Right) -> Left+Right end % Combine
  ).
```

# Scan in Erlang

- Remarkably like reduce.
- Reduce has
  - an upward pass to compute the grand total
  - a downward pass to broadcast the grand total.
- Scan has
  - an upward pass where the grand total – **just like reduce**
  - On the downward pass, we compute the total of all elements to the left of each subtree.
- Get the code at

http://www.ugrad.cs.ubc.ca/~cs418/2016-2/lecture/01-16/code/scan.erl

# The Scan Pattern

- It's a parallel version of *mapfold*, e.g. `lists:mapfoldl` and `lists:mapfoldr`.
- `wtree:scan(`*Leaf1, Leaf2, Combine, Acc0*`)`
  - *Leaf1*`(`*ProcState*`)` `->` *Value*
    Each worker process computes its *Value* based on its *ProcState*.
  - *Combine*`(`*Left*`,` *Right*`)` `->` *Value*
    Combine values from sub-trees.
  - *Leaf2*`(`*ProcState*`,` *AccIn*`)` `->` *ProcState*
    Each worker updates its state using the *AccIn* value – i.e. the accumulated value of everything to the worker's "left".
  - *Acc0*: The value to use for *AccIn* for the leftmost nodes in the tree.

# Scan example: prefix sum

```
prefix_sum_par(W, Key1, Key2) ->
  wtree:scan(W,
    fun(ProcState) -> % Leaf1
      lists:sum(wtree:get(ProcState, Key1)) end,
    fun(ProcState, AccIn) -> % Leaf2
      wtree:put(ProcState, Key2,
        prefix_sum(wtree:get(ProcState, Key1), AccIn)
      ) end,
    fun(Left, Right) -> % Combine
      Left + Right end,
    0 % Acc0
  ).

prefix_sum(L, Acc0) ->
  element(1,
    lists:mapfoldl(fun(X, Y) -> Sum = X+Y, {Sum,Sum} end,
                   Acc0, L)).
```

# More Examples of scan

- Account balance with interest:
  - Input: a list of transactions, where each transaction can be a deposit (add an amount to the balance), a withdrawal (subtract an amount from the balance), or interest (multiply the balance by an amount). For example:

    ```
    [{deposit, 100.00}, {withdraw, 5.43}, {withdraw, 27.75},
    ```

  - Output: the account balance after each transaction. For example, if we assume a starting balance of $1000.00 in the previous example, we get

    ```
    [1100.00, 1094.57, 1066.82, 1067.40, ...]
    ```

- Delete 3s
  - Given a list that is distributed across *NProc* processes, delete all 3s, and rebalance the list so each process has roughly the same length sublisth.
  - Solution (sketch):
    - Using scan, each process determines how many 3s preceed its segment, the total list length preceeding it, and the total list length after deleting 3s.
    - Each process deletes its 3s and send portions of its lists

# CS418 library vs. Lin & Snyder

- Top-down or bottom up?
  - Course library:
    - Master process initiates reduce or scan.
    - The `Leaf` and `Combine` functions are propagated down the tree.
    - Tallies are propagated up the tree, and the grand total is delivered to the master.
    - For scan, the "total of everything to the left" is propagated down the tree, and each worker process updates its local `ProcState`.
  - Lin & Snyder
    - The workers initiate reduce or scan.
    - Tallies are propagated up the tree.
    - Totals are propagated down the tree
      - Reduce: everyone gets the grand total
      - Scan: everyone gets the total of everything to the left

# Which is better?

- Lin & Snyder:
  - Better suited for writing real, parallel applications
  - In real applications, worker processes perform many operations, occasionally coordinating using reduce, scan, or similar operations.
  - Lin & Snyder avoid the bottleneck of a master process that dispatches tasks.
- The course library
  - It's implemented and it works.
  - Easier for simple examples, especially when making timing measurements.
    - We can start and stop our "stopwatch" at the master.
    - Avoids some details of how Erlang reports the current "time".
  - Allows for optimizations at the leaves that Lin & Synder don't
    - Lin & Snyder just take the combine operator and perform the combine in the obvious way a the leaves.
    - This ignores optimizations. Consider find the $k^{\text{th}}$ largest