CpSc 418

# Homework 2

Due: Feb. 1, 2017, 11:59pm
Early Bird: Jan. 30, 2017, 11:59pm

## 100 points.

Please submit your solution using the handin program. Submit your solution as
cs418 hw2
Your submission should consist of three files:

- hw2.erl: Erlang source code for the coding parts your solution.
- hw2.pdf or hw2.txt: PDF or plain-text for the written response parts of your solution
- hw2_test.erl: EUnit tests for your solution. See question 6.

Templates for hw2.erl and hw2_test.erl are available at
http://www.ugrad.cs.ubc.ca/~cs418/2016-2/hw/2/code.html.
The tests in hw2_test.erl are not exhaustive. If your code doesn't work with these, it will almost certainly have problems with the test cases used for grading. The actual grading will include other test cases as well.

Please submit code that compiles without errors or warnings. If your code does not compile, we might give you zero points on all of the programming problems. If we fix your code to make it compile, we will take off lots of points for that service. If your code generates compiler warnings, we will take off points for that as well, but not as many as for code that doesn't compile successfully. See the comments about errors, warnings, and guards at the end.

1. mean(WTree, DataKey), **(20 points)**.
WTree is a tree of worker processes from wtree:create, and DataKey is the key associated with a list of numbers that has been distributed across these processes. In other words, each process has a separate segment of the whole list. mean(WTree, DataKey) should compute the average of the elements in DataKey using wtree:reduce. You need to provide the Leaf, Combine, and possibly Root functions. For example, once you have implemented hw2:mean, you should be able to run the following sequence of commands in the Erlang shell:

```
1> W = wtree:create(8).  % create a tree of eight worker processes
[<0.58.0>,<0.59.0>,<0.60.0>,<0.61.0>,<0.62.0>,<0.63.0>,<0.64.0>,<0.65.0>]
2> L0_1000 = lists:seq(0,1000), ok.
ok % I added , ok. to prevent printing a lot of output.
3> workers:update(W, data, misc:cut(L0_1000, W)).
ok 4> Mean = hw2:mean(W, data).  500.0
```

The function misc:cut(L0_1000, W) creates a list of (length(W)) lists, i.e. [L1,L2,L3,L4,L5,L6,L7,L8] where
L0_1000 =:= L1++L2++L3++L4++L5++L6++L7++L8
and each of the lists, L1, ..., L8 has length(L0_1000) div 8 elements, or that many plus one (to handle lists whose length are not multiples of length(W)).

Implement mean(WTree, DataKey). I have some simple test cases in hw2_test.erl; you should add some more (see Question 6).

2. vec_mean(WTree, DataKey), **(20 points)**.
This is similar to the function mean from Question 1. WTree is a tree of worker processes from wtree:create, and DataKey is associated with a list of vectors that has been distributed across the processes of WTree. Each vector is represented as a list of numbers, and all of the vectors should have the same length. Your code is allowed to fail if the value associated with DataKey does not satisfy

these properties[1]. `vec_mean(WTree, DataKey)` should compute the average of the vectors. For example, once you have implemented `hw2:vec_mean`, you should be able to run the following sequence of commands in the Erlang shell:

```
5> W = wtree:create(8).  % create a tree of eight worker processes
[<0.58.0>,<0.59.0>,<0.60.0>,<0.61.0>,<0.62.0>,<0.63.0>,<0.64.0>,<0.65.0>]
6> VList = [hw2:vec_add(N, [1,2,3]) || N <- lists:seq(0,1000)], ok.
ok % I added , ok. to prevent printing a lot of output.
7> workers:update(W, vdata, misc:cut(VList, W)).
ok
8> VMean = hw2:vec_mean(W, vdata).
[501.0,502.0,503.0]
```

Implement `vec_mean(WTree, DataKey)`. Of course, you should do this with a reduce or scan using `wtree:reduce` or `wtree:scan` respectively. I have some simple test cases in `hw2_test.erl`; you should add some more (see Question 6).

3. `set_nth(N, Fun, List)`, **(10 points)**.
   If $X$ is the `N`th element of `List`, `set_nth(N, Fun, List)` returns the list obtained by replacing the `N`th element of `List` with `Fun(`$X$`)`. For example:
   `set_nth(3, fun(X) -> X*X+2 end, [1,4,9,16]) -> [1,4,83,16].`
   Your function should fail with an error if the parameters aren't reasonable, e.g. `N` is not an integer, `N =< 0`, etc.

   Implement `set_nth` and add suitable test cases to `hw2_test.erl`. **Note:** `set_nth` is *sequential*; you shouldn't use reduce, scan, spawn, or any other parallel stuff.

4. `bank_statement(WTree, SrcKey, DstKey, InitialBalance)`, **(20 points)**.
   `WTree` is a tree of worker processes from `wtree:create`; `SrcKey` is the key for a list of "transactions" (described below); `DstKey` is the key for a list with the balance after each transaction; and `InitialBalance` is the initial balance of the account. The list associated with `SrcKey` should be distributed across the processes of `WTree` (for example, using `workers:update` as in Questions 1 and 2). The list associated with `DstKey` is a list of number that is created by executing `bank_statement`.

   The transaction list is a list of tuples of the form `{Operation, Value}`. The three operations are described below:

   `{deposit, Amount}`: `Amount` is added to the current balance. `Amount` must be a number.

   `{withdraw, Amount}`: `Amount` is subtracted to the current balance. `Amount` must be a number.

   `{interest, Rate}`: The current balance is multiplied by $1 + \frac{\texttt{Rate}}{100}$.

   Table 1 shows an example.

   Implement `bank_statement` and add suitable test cases to `hw2_test.erl`. Of course, you should do this with a reduce or scan using `wtree:reduce` or `wtree:scan` respectively.

5. `sliding_average(WTree, SrcKey, DstKey, Kernel, InitialPrefix)`, **(25 points)**.
   `WTree` is a tree of worker processes from `wtree:create`; `SrcKey` is the key associated with a list of numbers that has been distributed across these processes – we will call this list `SrcList` in the rest of the problem description. `Kernel` is a list of numbers that gives the weights for the sliding average –

[1]In case of invalid data for a worker process, the worker process should fail with an error. Often when one worker fails, they all fail. The code in `wtree` catches these errors and puts together an error report. The report is often lengthy and repetitive (because each process failed). However, you can determine what kind of error occurred and the source code line at which the error occurred by reading the report. I've tried to keep errors from killing the worker processes. Thus, you can use `workers:retrieve` or other functions in the `workers` or `wtree` modules to figure out what happened.

Table 1: Bank statement example (for Question 4)

| Transaction | Balance after transaction |
|---|---|
| Initial Balance | 0.00 |
| `{deposit, 100.00}` | 100.00 |
| `{deposit, 500.00}` | 600.00 |
| `{withdraw, 50.00}` | 550.00 |
| `{withdraw, 25.00}` | 525.00 |
| `{withdraw, 14.00}` | 511.00 |
| `{withdraw, 11.00}` | 500.00 |
| `{interest, 5.00}` | 525.00 |
| `{withdraw, 17.00}` | 508.00 |
| `{deposit, 42.00}` | 550.00 |
| `{interest, -3.00}` | 533.50 |
| `{withdraw, 123.45}` | 410.05 |
| `{deposit, 19.95}` | 430.00 |
| `{interest, 2.33}` | 440.02 |
| `{deposit, -0.33}` | 439.69 |
| `{withdraw, 192.68}` | 247.01 |
| `{withdraw, 300.00}` | -52.99 |
| `{interest, 20.00}` | -63.59 |
| `{deposit, 10.00}` | -53.59 |
| `{interest, -5.00}` | -50.91 |
| `{withdraw, 14.00}` | -64.91 |
| `{deposit, 100.00}` | 35.09 |

```
Kernel = [0.25, 0.25, 0.5].
SrcList = [1, 2, 10, 12, 14, 2, 2, 2, 0, -1, -8, 14].
InitialPrefix = [0, 0],
PreSrc = [0, 0, 1, 2, 10, 12, 14, 2, 2, 2, 0, -1, -8, 14].
% Compute the first element of DstList
% [0,    0,    1,    2,    10,   12, 14, 2, 2, 2, 0, -1, -8, 14] = PreSrc
% [0.25, 0.25, 0.5] = Kernel
% [0,    0,    0.5] = Kernel*PreSrc (at positions of overlap)
% 0.5 = sum(Kernel*PreSrc) = first element of DstList
% Now, move the kernel one position to the right
% and compute the second element of DstList
% [0,    0,    1,    2,    10,   12, 14, 2, 2, 2, 0, -1, -8, 14] = PreSrc
%        [0.25,0.25, 0.5] = Kernel
%        [0,   0.25, 1] = Kernel*PreSrc (at positions of overlap)
% 1.25 = sum(Kernel*PreSrc) = second element of DstList
% we continue shifting Kernel to the right along PreSrc
% and compute the remaining values of DstList
DstList = [0.5, 1.25, 5.75, 9.0, 12.5, 7.5, 5.0, 2.0, 1.0, 0.0, -4.25, 4.75].
% stop when we reach the end of DstList.
```

Figure 1: Computing a sliding average

typically, the elements of `Kernel` will sum to 1, but we won't require that. `InitialPrefix` is a list of `length(Kernel)-1` numbers that we treat as if they came before `SrcLst`. The goal is to compute a "weighted, sliding average" for each value in the list associated with `SrcKey`. The `sliding_average` computation will create an "average" for each element of `SrcList` we will call this list averages `DstList` in the rest of the problem description. The list `DstList` will be associated with the key `DstKey` at the end of the `sliding_average` computation.

Let `K = length(Kernel)` and `PreSrc = InitialPrefix++SrcList`. The formula for `DstList` is

$$\texttt{lists:nth(N, DstList)} = \sum_{I=1}^{\texttt{K}} \texttt{lists:nth(I, Kernel)} * \texttt{lists:nth(N+I, PreSrc)}$$

This really needs an illustration – see Figure 1. I've also provided a sequential implementation; see `sliding_average_seq` in the `hw2.erl` template.

6. **Test cases (5 points):** The test case file distributed with this homework, `hw2_test.erl`, provides a fairly minimal set of EUnit tests. You should definitely add more tests and run them before you submit your solutions to the other problems. Include your version of `hw2_test.erl` with your solution.

   If we find errors in your code with our tests, we will scrutinize your test cases and take off points if you hadn't been careful in getting good test coverage. As with homework 1, the collaboration policy is relaxed *just for* `hw2_test.erl`. If you collaborate with others, you can jointly write one `hw2_test.erl` and all share it. **Of course,** everyone in your collaboration group must submit a copy of the file, and you each must clearly state who your collaborators were in a comment within the first 10 lines of your `hw2_test.erl`.

## Why?

**Question 1**: This is supposed to be a simple example of reduce. Try it right away. If you find it easy, you can be happy and go to the other problems. If you get stuck, then make sure you ask for help as early as possible – we have office hours and tutorials.

**Question 2**: This is a generalization of the previous question, and another step on our path to a parallel implementation of the $k$-means clustering algorithm.

**Question 3**: I thought we'd take a break from all of this map and reduce stuff, and do something with a higher order function. Like Question 1, this is supposed to be fairly basic. If you get stuck, ask for help right away.

**Question 4**: This is a reduce or scan problem (which is it?) that is intended to make you think a bit about how to make an operation "associative".

**Question 5**: This is intended as a more challenging reduce or scan. I was considering including a part where you would figure out an asymptotic formula for the run time, measure the actual run time, compare the two, and calculate speed-ups. However, this homework set seems challenging enough already; so, I'll save those questions for later.