CpSc 418

Homework 2

Due: Jan. 31, 2018, 11:59pm
Early Bird: Jan. 29, 2018, 11:59pm

73 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` PDF for the written response parts of your solution and the plots.
- `hw2_tests.erl`: EUnit tests for your solution.

Templates for `hw2.erl` and `hw2_tests.erl` are available at
`http://www.ugrad.cs.ubc.ca/~cs418/2017-2/hw/2/code.html`.

The tests in `hw2_tests.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.

We will take off points for code that prints results unless we specifically asked for print-out. For this assignment, the functions you write should return the specified values, but they should not print anything to `stdout`. Using lists:format when debugging is great, but you need to delete or comment-out such calls before submitting your solution. Printing an error message to `stdout` when your function is called with invalid arguments is acceptable but not required. Your code must fail with some kind of error when called with invalid arguments.

1. **Poetry Jam** (28 points)
   I was delighted when Homer, Li Bai, William Shakespeare, and Donald Rumsfeld agreed to participate in the CPSC 418 poetry jam. They were all very eager to recite their poems and the result was a rather scrambled recitation. You can try executing

   ```
   hw2:poetry_jam().
   ```

   to read a transcript of this fiasco. We need locks so that the poets will give their recitations one at a time. For this problem, you will use Erlang processes and messages to implement locks.

   (a) Implement Erlang functions to implement a lock process. (**12 points**)

   i. Implement a function, `lock_create()` that spawns a lock process and returns its pid. We'll call this pid `LockPid` in the remaineder of this question. A lock process is a Finite State Machine with two states: `locked` and `unlocked`. See Rage Against the Finite State Machines – – What are They? *LYSE* to get a description of how to implement a finite-state machine. You can do this one brute force by writing `locked` and `unlocked` functions. You don't need to figure out how to use `gen_fsm` which is mentioned at the end of the What are They? section (and described in the rest of the Rage chapter). The lock process should be initially in the unlocked state.

   ii. When the lock process is in the unlocked state, it receives messages of the form
   `{Pid, acquire}`

which is a request by `Pid` to acquire the lock. The lock process responds by sending the message

```
{LockPid, granted}
```

back to `Pid`. The lock process then transitions to the locked state. Note that it will need to keep track of `Pid` as the "owner" of the lock.

iii. When the lock process is in the locked state, it receives messages of the form

```
{OwnerPid, release}
```

where `OwnerPid` is the current owner of the lock. The lock then transitions back to the unlocked state. Other messages should be ignored – in particular, `acquire` and `exit` messages remain in the in-box to processed when the lock is in the unlocked state.

iv. When the lock process is in the unlocked state, it can also receive a message of the form

```
exit
```

The lock process exits in response to such a message.

(b) Secret messages (**12** points)
Review We love messages but keep them secret, in the More on Multiprocessing chapter of *LYSE*. The lock interface should provide *functions* for interacting with the lock. You wrote `lock_create` above. Now implement:

i. `lock_acquire(Lock)` This function handles the exchanging of messages with `Lock` to handle acquiring the lock. This function should not return until this process has acquired the lock.

ii. `lock_release(Lock)` This function handles the exchanging of messages with `Lock` to handle releasing the lock. Note that this function can return as soon as it has sent the release message to `Lock`. That ensures that the lock will eventually be unlocked, which is all we require.

iii. `lock_destroy(Lock)` Terminate the `Lock` process, thus freeing its resources.

(c) Test your lock (**4 points**)
Run the `hw2:poetry_jam()` function. If your lock is correct, all four poets should recite their poems without interrupting each other. Furthermore, your lock process should terminate when `hw2:poetry_jam` calls `lock_destroy` – we'll check to make sure there aren't any extra processes loitering around (a few milliseconds) after `hw2:poetry_jam` returns. Read the poems; choose your favorite, and complete the function `hw2:favorite_poet()`.

2. **We All Have Our Moments** (20 points)
Moments are a useful way of quantifying properies of random distributions and can be important for "big data" problems – I figured you should see them before you graduate. ☺

The k^{th} *central moment* of a random variable, X, is

$$\mu_k \quad = \quad E\left[(X - E[X])^k\right]$$

where $E[X]$ is the *expected value* (i.e. mean or average) of X.

If we have a list of numbers, $x_1, x_2, \ldots, x_N$ that are samples from the distribution of X, we can estimate the mean and the moments with

$$\begin{array}{rcl} E[X] & \approx & \frac{1}{N}\sum_{i=1}^{N} x_i \\ \mu_k & \approx & \frac{1}{N}\sum_{i=1}^{N}(x_i - E[X])^k \end{array}$$

I wrote $\approx$ because if we have a finite sample, we only get an estimate, but the estimate gets better for larger N. If I were a real statistician, I would make other corrections as well, for example, dividing by $N-1$ instead of N when estimating μ_2 the variance, but I'm not sure what the correction is for arbitrary k; so, I'll just use $1/N$ for all of them to keep it simple.

The template file, `hw2.erl`, includes a function `hw2:moment(X, M, X0)` that estimates the $\texttt{M}^{\text{th}}$ central moment of `X` assuming that `X0` is the mean of `X`. In this problem, you will write a parallel implementation, `moment_par(W, Key, M, X0)` using `wtree:reduce`.

(a) Implement `mp_leaf(List, M, X0) -> {LengthList, SumListM.}` (**2 points**)

- `List` is a list of numbers.
- `M` is a number, typically a positive integer.
- `X0` is a number, typically the expected value of the distribution sampled by `List`.
- `LengthList` is the length of `List`.
- `SumListM` is the sum of $\texttt{D}^{\texttt{M}}$ for $\texttt{D} = \texttt{X} - \texttt{X0}$ for each element `X` of `List`.

(b) Implement `mp_combine(Left, Right) -> SubtreeSummary.` (**2 points**)
`Left` and `Right` are the summaries for the left and right subtrees of this vertex in the tree, and `SubtreeSummary` is the summary for the whole subtree. `Left`, `Right`, and `SubtreeSummary` should have the same form as the values returned by `mp_leaf`, i.e. tuples of the form `{Length, SumM}`.

(c) Implement `mp_Root(RootSummary) -> MomentM.` (**2 points**)
`RootSummary` should have the same form as the values returned by `mp_leaf` and `mp_combine`.

(d) Test your code. (**4 points**)
Add test cases to `hw2_tests.erl` for `mp_leaf`, `mp_combine`, `mp_root`, and `moment_par`. Note that `moment_par` should work correctly once you've implemented `mp_leaf`, `mp_combine`, and `mp_root`. The function `moment_test` in `hw2.erl` provides many ways to interactively test `moment_par` from the Erlang shell. For example,

```
1> hw2:moment_test(8, hw2:erlang(100000,3,2.45), 3, 3/2.45).
moment_test passed:  N_Data = 100000,
MomentPar = 4.1768203047770e-1, MomentSeq = 4.1768203047770e-1
2>
```

You can use ideas or code-fragments from the code provided in `hw2.erl` to help write your tests for `hw2_tests.erl`.

(e) Speed up versus number of processors (**4 points**)
Plot the speed-up versus the number of processors when running on `thetis.ugrad.cs.ubc.ca`. Use a list of length 1,000,000 for your tests. When running on `N_Procs` processors, the list should be evenly distributed across the processors, for example by using `workers:rlist`. The function `hw2:moment_time(N_Procs, N_Data, M)` gives an example of how to do this, and you are welcome to use `hw2:moment_time` to make your measurements. Note that `hw2:moment_time(0, N_Data, M)` reports the time for the *sequential* computation using `hw2:moment(List, M)`. Include data points for $\texttt{N\_Procs} \in \{2, 4, 8, 16, 32, 64, 128\}$ – you can add more if it helps show the trends more clearly. Use a semi-log plot – i.e. `N_Procs` on a log scale and *SpeedUp* on a linear scale.

(f) Speed up versus length of the list (**4 points**)
Plot the speed-up versus the length of the list when running on `thetis.ugrad.cs.ubc.ca`. Use $\texttt{N\_Procs} = 64$ for the parallel case and $\texttt{N\_Data} \in \{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}\}$. Use a semi-log plot – i.e. `N_Data` on a log scale and *SpeedUp* on a linear scale. For the sequential case, here are the times when $\texttt{N\_Data} \geq 2^{20}$

`N_Data`	time (seconds)
2^{20}	0.214
2^{21}	0.404
2^{22}	0.793
2^{23}	1.601
2^{24}	3.345

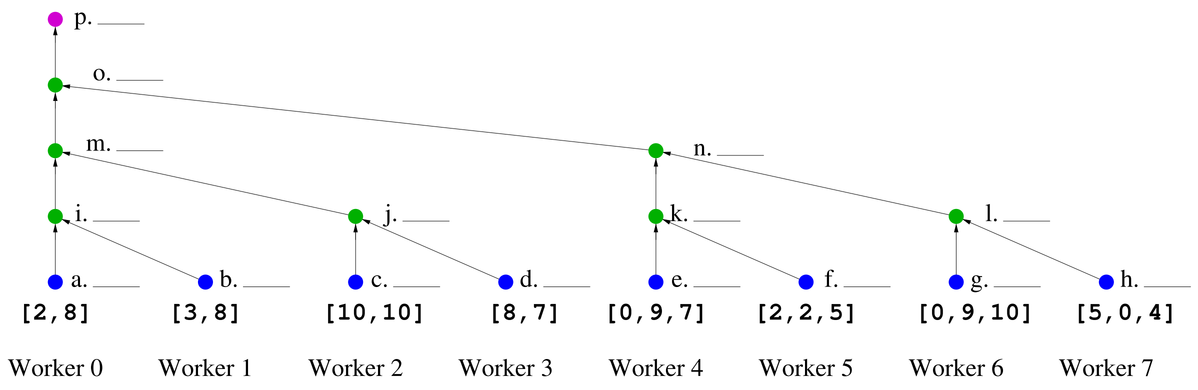

Figure 1: A process tree for scan

I'm providing these numbers so we don't have 80 students running CPU intensive timing measurements right before the deadline. The parallel versions should all run in less than one-second per run, and thus take about one second per measurement with `time_it:t/1`.

(g) Observations (**2 points**)
Write a few observations about the speed-ups you observe. Do your observations match the models that have been presenting in class and the readings, or do you see discrepancies. A short answer of two to five sentences should suffice.

3. **Scan Tree** (10 points)
Figure 1 shows a tree for computing a scan. The list

```
[2,8,3,8,10,10,8,7,0,9,7,2,2,5,0,9,10,5,0,4]
```

is distributed across the eight workers. These numbers are integers chosen uniformly at random from $[0, 10]$. Thus, the expected value for such an integer is 5. We are interested in computing the cumulative second moment of this list. In question 4, you will implement this using `wtree:scan`. This problem is a warm-up where you will determine the intermediate results of the scan computation.

(a) The upward pass (**5** points)
Complete the function `q3a` in `hw2.erl` to indicate the results at each step of the upward pass (i.e. the "reduce" phase). For example, the tuple `{b, {2, 13}}` indicates that worker process 1 computes `{2, 13}` as the result of its `Leaf` computation. In a bit more detail, worker process 1 has the sublist `[3,8]`, the length of this list is `2`. The sum of the squares of the differences of the list elements from the expected value is:

$$(3-5)^2 + (8-5)^2 \quad = \quad 13$$

Likewise, the tuple `{j, {4, 63.0}}` indicates that the summary for the workers 2 and 3 together have 4 leaves and the sum of the squares of the difference of their list elements from the expected value is 63.

(b) The downward pass (**5** points)
Complete the function `q3b` in `hw2.erl` to indicate the results at each step of the downward pass of the scan. Each node receives from its parent the summary for all leaves to the left of the subtree rooted at that node. For example, the tuple `{j, {4, 31}}` indicates that there are four leaves to the left of the subtree rooted at `j`, and the sum of the squares of the difference of the values of

these leaves and the expected value is 31. If you want even more detail, the subtree at j consists of workers 2 and 3. The leaves to the left of node j are the elements of the list segments held by workers 0 and 1. These leaves are `[2,8,3,8]`. There are four such leaves, and

$$(2-5)^2 + (8-5)^2 + (3-5)^2 + (8-5)^2 \quad = \quad 31$$

For the tuple `q, your_answer([], 'q3a.a')`, you need to write the result list that is computed by worker 3. Use an inclusive scan. For example, the result list computed by worker 6 is:

```
[12.133333333333333,12.375,13.117647058823529]
```

4. **Scanning Moments** (15 points)

   (a) Implement `moment_fold(List, M, X0) -> List2` (**5 points**)
   Compute the cumulative `M`th moment for `List`, where `X0` is the expected value of elements in `List`. This means that for $1 \leq$ `N` $\leq$ `length(List)`,

$$\texttt{lists:nth(N, List2)} \quad = \quad \frac{1}{N}\sum_{I=1}^{N}(\texttt{lists:nth(I, List)} - \texttt{X0})^{\texttt{M}}$$

   For example,

```
moment_fold([1,3,14,-2], 1, 0) -> [1.0,2.0,6.0,4.0].   % running average
moment_fold([1,3,14,-2], 2, 4) -> [9.0,5.0,36+(2/3),36.5].   % running variance
```

   (b) Implement `moment_scan(W, KeySrc, KeyDst, M, X0) -> Est_Moment` (**5 points**)
   Compute the cumulative `M`th moment for the list that is distributed over the workers of `W` and associated with `KeySrc`. In a bit more detail, if `List` is distributed over the workers of `W` with key `KeySrc`, then `Est_Moment` should be the same as

```
moment(W, KeySrc, M, X0)
```

   (we'll accept "close" in the sense of `hw2:close/2`). Furthermore, after executing

```
moment_scan(W, KeySrc, KeyDst, M, X0)
```

   The workers of `W` should have a list associated with `KeyDst` is the list of cumulative `M`th moments of `List`. Figure 2 shows an example.

   (c) Tests and Efficiency (**5 points**)
   Add some test cases for `moment_fold` and `moment_scan`. `hw2_tests.erl` Your implementations should have reasonable efficiency.

Why?

I'll summarize the learning objectives connected with each question.

```
2> W = wtree:create(8).
[<0.63,0>,<0.64,0>,<0.65,0>,<0.66,0>,<0.67,0>,<0.68,0>,<0.69,0>,<0.70,0>]
3> R = [X-1 || X <- misc:rlist(20, 11)].
[1,2,1,6,5,8,0,9,4,10,1,10,10,2,9,0,9,1,8,1]
4> workers:update(W, src, misc:cut(R, W)).
ok
5> workers:retrieve(W, src).
[[1,2],[1,6],[5,8],[0,9],[4,10,1],[10,10,2],[9,0,9],[1,8,1]].
% The previous line shows how R was distributed over the workers of W.
6> hw2:moment_scan(W, src, dst, 2, 5.0).  % cumulative variance
14.55
7> hw2:moment(R, 2, 5.0).
14.55 % good, they agree
8> R2 = workers:retrieve(W, dst).
[[16.0,12.5],                                         % the segment on Worker 0
 [13.666666666666666,10.5],                           % the segment on Worker 1
 [8.4,8.5],                                           % the segment on Worker 2
 [10.857142857142858,11.5],                           % the segment on Worker 3
 [10.333333333333334,11.8,12.181818181818182],        % the segment on Worker 4
 [13.25,14.153846153846153,13.785714285714286],       % the segment on Worker 5
 [13.933333333333334,14.625,14.705882352941176],      % the segment on Worker 6
 [14.777777777777779,14.473684210526315,14.55]]       % the segment on Worker 7
9> hw2:close(lists:flatten(R2), hw2:moment_fold(R, 2, 5)).
true % Yay!
```

Figure 2: `moment_scan` example