CpSc 418 **Homework 3** Due: October 20, 2017, 11:59pm
Early Bird: October 18, 2017, 11:59pm

117 points.

Please submit your solution using the handin program. Submit your solution as
cs418 hw3
Your submission should consist of three files:

- `hw3.erl`: Erlang source code for the coding parts your solution.
- `hw3.pdf` or `hw3.txt`: PDF or plain-text for the written response parts of your solution.
- `hw3_tests.erl`: EUnit tests for your solution.

A template for `hw3.erl` is available at
http://www.ugrad.cs.ubc.ca/~cs418/2017-1/hw/3/src/hw3.erl
Please submit code that compiles without errors or warnings. If your code does not compile, we will probably 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.

The Questions

1. **Biggest Distance** (35 points)

   In this problem you will practice your reduce design skills with less scaffolding than in the previous homework. Given a list of vectors (all of identical dimension), your task is to find the pair of consecutive vectors with the greatest distance between them and the index of the first of those two vectors. We will use the standard Euclidean (2-norm) distance between two vectors in dimension d:

$$\text{dist}(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}.$$

   (a) (2 points) Distance between two vectors.
   We will use an Erlang list of length d to store each vector. The function `distance(ListX, ListY)` has already been written for you using some functions from the `lists` and `math` modules. You need to complete the `distance_combine(Xi, Yi)` function. You will find a `distance_examples()` function which provides some sample input and output for `distance()`. Your implementation of `distance_combine()` should be $\mathcal{O}(1)$ time so that `distance()` is $\mathcal{O}(d)$ time.

   (b) (6 points) Serial biggest distance.
   Write the code for the function

   ```
   biggest_distance_ser(List) -> {Distance, Index}.
   ```

   where `List` is a list of vectors (each vector is a list, and all vectors are the same length), *Distance* is a non-negative float specifying the distance between the pair of consecutive vectors which are furthest apart, and *Index* is an integer specifying the index of the first of those vectors. If `List` is empty or contains only one vector, return `{none, none}`. If multiple pairs of vectors have exactly the same distance between them, return in the index to the first vector in the first pair.

The function `biggest_distance()` is very simple, but you also must complete the helper `bds()`. You will probably find the function `max_distance_tuple()` helpful when completing these functions. The function `biggest_distance_ser_examples()` provides some sample input and output. For an input list with N vectors, your implementation should make $\mathcal{O}(N)$ calls to `distance()` for a total time of $\mathcal{O}(Nd)$.

(c) (4 points) Parallelizing biggest distance: segment summaries.

As discussed in class and in the previous homework, a key step in parallelizing an algorithm using reduce (or scan) is figuring out how to summarize the information in each subtree. This summary will be created by the leaf nodes, combined by the interior nodes, and then all or some of it will be reported by the root as the result of the reduce.

Clearly our sublist summary will contain the `{distance, index}` tuple reporting the biggest distance that we have seen between any two pairs of vectors in the sublist. What other information will we need in order to combine the summaries for two sublists? Briefly justify your choice(s). Specify the data structure which will store your summary and the types of all of its elements.

Hint: Summaries often contain information about

- The left end of a sublist.
- The whole sublist (such as the `{distance, index}` tuple, but not necessarily just that).
- The right end of a sublist.

(d) (5 points) Parallelizing biggest distance: leaf nodes.

Write `bdp_leaf(SubList)`, which returns the summary information discussed in the previous part.

(e) (8 points) Parallelizing biggest distance: combine nodes.

Write `bdp_combine(Left, Right)`, which combines two summaries into one. Don't forget that any indexes coming from the right summary must be shifted to account for the length of the left sublist. You may find the function `max_distance_tuple()` useful.

(f) (2 points) Parallelizing biggest distance: root node.

Write `bdp_root(RootSummary)`, which extracts the appropriate return value from the summary of the entire list.

(g) (3 points) Parallelizing biggest distance: calling reduce.

Write the code for the function

```
biggest_distance_par(W, ListKey) -> {Distance, Index}.
```

where `W` is a worker tree on which to perform the reduce and `ListKey` is the key under which each worker has its local sublist stored. The return value is the same as for the serial version.

Hint: This part should just require filling in our reduce template call to `wtree:reduce()` with the appropriate functions you designed above. See the September 22 lecture slides for an example of the reduce template.

(h) (5 points) Add test cases to your `hw3_tests` module to test both the serial and parallel versions. Your test cases should include corner cases—typically ones where you run your code with a small number of processes and some short lists to make sure that the various cases in your functions are working correctly. You may find the function `random_vector_list()` (or modifications thereof) useful when creating your test cases.

2. **Running Counts** (55 points)

In this problem you will explore the scan pattern using the course library. Given a list of Erlang elements, define the *histogram count* for that list as a mapping from unique elements in the list to the number of times that element appears in the list. An element can be of any Erlang type, and the list may contain elements of mixed types. Consider the following examples:

- A list of atoms `[ alan, george, alan, ian, mark, alan, ian, paul ]` has histogram count:

  `alan => 3, george => 1, mark => 1, ian => 2, paul => 1`

  where we use the notation *ListElement* `=>` *Count* to denote that the *ListElement* has appeared in the list *Count* times. Note that the *ListElement* `=>` *Count* pairs may appear in any order in the mapping.
- A list of integers `[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 2, 4, 6, 8, 0, 3, 6, 9, 0 ]` has histogram count:

  `0 => 4, 6 => 3, 2 => 2, 3 => 2, 4 => 2, 8 => 2, 9 => 2, 1 => 1, 5 => 1, 7 => 1`

  where the pairs are ordered from largest to smallest count for reading convenience.
- A list of mixed types `[ "AA", 42, poodle, 3.14159, "BBBB", "AA", "AA", 42, "AA", true ]` has histogram count:

  `[ 65, 65 ] => 4, 42 => 2, [ 66, 66, 66, 66 ] => 1,`
  `true => 1, poodle => 1, 3.14159 => 1`

  where Erlang has reported the strings in their list format (`[ 65, 65 ]` = `"AA"` and `[ 66, 66, 66, 66 ]` = `"BBBB"`).

We will store histogram counts using the Erlang `maps` data type. Although not discussed in the main chapters of LYSE, as of Erlang OTP 17.0 `maps` is the preferred manner of storing *Key*/*Value* data structures[1] and is intended to replace the other Key/Value stores discussed in LYSE A Short Visit to Common Data Structures, such as the `orddict`, `dict` and `gb_trees` types. You can read more about the fancy features of `maps` in the LYSE Postscript: Maps, but we will not need any of the new syntax or pattern matching features that `maps` come with. Instead, we will just use the functional interface provided through the `maps` module. In particular, that module provides functions for creating empty maps; converting maps to and from lists; getting and setting values given keys; and iterating over the keys, values or key/value tuples.

Finding the final histogram count for a list distributed across a set of workers makes for a nice parallel reduce problem, but for this question we will solve a version of the problem that requires a scan. Given a list of Erlang elements, the *running histogram count* is a list of the same length where element i of the output list is a histogram count (as defined above) for the prefix of the input list from element 1 to element i. For the examples listed above:

- The list of atoms `[ alan, george, alan, ian, mark, alan, ian, paul ]` has running histogram count:

input list element	output list count at element i							
`alan =>`	1	1	2	2	2	3	3	3
`george =>`	-	1	1	1	1	1	1	1
`ian =>`	-	-	-	1	1	1	2	2
`mark =>`	-	-	-	-	1	1	1	1
`paul =>`	-	-	-	-	-	-	-	1

  where a "`-`" symbol indicates that the corresponding element has not yet appeared in the input list and hence it will not appear in the histogram count mapping for that index; for example, the output list's first element will be a map containing only the *Key* `alan` and the *Value* 1, while the atom `paul` will not appear as a *Key* in any of the output list's maps until the final one.
- The list of integers `[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 2, 4, 6, 8, 0, 3, 6, 9, 0 ]` has histogram count:

[1] Such data structures are common in many languages, including the "map" interface in Java (often instantiated with the HashMap class) or the "dictionary" data structure in Python.

input list element	output list count at element i																			
0 =>	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	3	3	3	3	4
1 =>	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2 =>	-	-	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2
3 =>	-	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
4 =>	-	-	-	-	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
5 =>	-	-	-	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6 =>	-	-	-	-	-	-	1	1	1	1	1	1	1	2	2	2	2	3	3	3
7 =>	-	-	-	-	-	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1
8 =>	-	-	-	-	-	-	-	-	1	1	1	1	1	1	2	2	2	2	2	2
9 =>	-	-	-	-	-	-	-	-	-	1	1	1	1	1	1	1	1	1	2	2

Additional examples of final and running histogram counts can be found in the file `hw3_scan_examples.txt` at the course web site.

As you work through this question, check in the template code comments for hints about what functions from the `maps` module might be useful. The functions `print_sorted_by_count()` and `print_running_count()` provided in the template file are designed to display a single histogram count and a running histogram count respectively, and may be useful for debugging.

(a) (6 points) Running count serial version.

Code for the function `running_count_ser()` has been provided in the template. Write the helper function `rcs()` so that it is tail recursive. The function `running_count_ser_examples()` can be used for preliminary testing. Example output is provided in `hw3_scan_examples.txt`.

(b) (3 points) Parallelizing running count: segment summary. At the end of the scan, each leaf node will need to compute the running histogram count at each element in its portion of the input list, but those counts will need to take into account all of the elements of the full input list to the left of this node (in other words, input elements whose index comes before the indexes in this node's portion of the list), even though those elements are not available on this node. What information does this node need to know about the elements of the input list to its left? Briefly justify your answer.

Note that the size of your summary should not depend (or at least not depend strongly) on the length of the list that it is summarizing; for example, it should not be the entire running histogram count output for that list.

(c) (3 points) Parallelizing running count: leaf nodes before upward pass.

Write

```
rcp_leaf1(SubList) -> Summary
```

where `SubList` is that portion of the input list stored on the current leaf node and `Summary` is the segment summary you determined above. Note that this function cannot modify the node's `ProcState`, so intermediate results will be discarded.

(d) (4 points) Parallelizing running count: combine nodes for upward and downward pass.

Write

```
rcp_combine(LeftOrParent, RightOrLeft) -> ParentOrRight
```

where all three arguments take the form of the summary information for this problem. During the upward pass `LeftOrParent` will be the summary from the left child, `RightOrLeft` will be the summary information from the right child, and `ParentOrRight` will be the summary information

passed to the parent node in the tree. During the downward pass `LeftOrParent` will be the summary information from the parent representing all nodes to the left of this subtree, `RightOrLeft` will be the summary information from the left child (remembered from the upward pass by the current node), and `ParentOrRight` will be the summary information passed to the right child. (During the doward pass the left child will receive `LeftOrParent` without modification.)

(e) (3 points) Parallelizing running count: root node.
Write

```
rcp_root(RootSummary) -> { ReturnValue, Acc0 }
```

where `RootSummary` is the result returned by the final `rcp_combine(Left, Right)` call of the upward pass through the tree; consequently, this summary takes into account the entire input list. The output `ReturnValue` is returned by the call to `wtree:scan()`. The downward pass is then started by passing `Acc0` to the root's left child and `rcp_combine(Acc0, Left)` to the root's right child (where `Left` is the summary which was passed up from the left child during the upward pass).

(f) (6 points) Parallelizing running count: leaf nodes after downward pass.
Write

```
rcp_leaf2(ProcState, AccIn, ListKeyIn, ListKeyOut) -> NewProcState
```

where `ProcState` is the local node's Key/Value state, `AccIn` is the summary information about all portions of the input list to the left of this node, `ListKeyIn` is the key to retrieve this node's portion of the input list, `ListKeyOut` is the key under which this node's portion of the output list should be stored, and `NewProcState` is the modified Key/Value state for this node which should include its portion of the output list.

As mentioned earlier, you may need to repeat some of the calculation from `rcp_leaf1()`, since that function was not able to store its intermediate results.

(g) (4 points) Parallelizing running count: calling scan.
Write the code for the function

```
running_count_par(W, ListKeyIn, ListKeyOut) -> HistogramCount.
```

where `W` is a worker tree on which to perform the scan, `ListKeyIn` is the key under which each worker has its local portion of the input list stored, `ListKeyOut` is the key under which each worker should store its portion of the output list, and `HistogramCount` is the histogram count summarizing the whole list stored as a single `maps` object (*not* the running histogram count).

The function `running_count_par_examples()` can be used for preliminary testing. Example output is provided in `hw3_scan_examples.txt`.

Hint: This part should just require filling in our scan template call to `wtree:scan()` with the appropriate functions you designed above. See the September 25 lecture slides for an example of the scan template.

(h) (5 points) Add test cases to your `hw3_tests` module to test both the serial and parallel versions. Your test cases should include corner cases—typically ones where you run your code with a small number of processes and some short lists to make sure that the various cases in your functions are working correctly. When designing your test cases, do not let the number of unique elements in your input lists grow too large—a few hundred is sufficient.

(i) (9 points) Theoretical analysis.
In our implementation of the histogram count problem, we chose to use the `maps` *Key*/*Value* data structure to store the counts for each element of the input list. A well designed implementation

of `maps` should require $\mathcal{O}(1)$ time to create, retrieve or replace a *Value* given a *Key*. If the input list has N elements, the number of unique elements in the input list is K, the number of workers used in the parallel case is P, and assuming that `maps` is well designed, what is the "big-O" time complexity of `running_count_ser()`, `rcp_leaf1()`, `rcp_combine()`, `rcp_root()`, `rcp_leaf2()` and `running_count_par()` in terms of N, K and P?

If the size of *Key*s and *Value*s is $\mathcal{O}(1)$, a well designed implementation of `maps` will require space $\mathcal{O}(K)$ to store K *Key*/*Value* pairs. Assuming this to be true, what is the "big-O" space required by `running_count_ser()`, the `ProcState` variable stored on each worker after calling `running_count_par()` (assuming an equal partition of the input list), and each message passed through the tree during the `wtree:scan()`?

(j) (12 points) Empirical analysis of run time.

Measure the execution time of `running_count_ser()` and `running_count_par()` on an input list generated by `misc:rlist(`N`, `M`)` (serial case) or `wtree:rlist(W, `N`, `M`, inputKey)` (parallel case) for each of the data sets listed below. Report the sequential time, the parallel time and the speedup. Run your experiments on `thetis.ugrad.cs.ubc.ca`. Do not include the time to generate the input list or (in the parallel case) retrieve the output list.

- Fix $P = 32$ and $M = 100$. Report the run-times and speedups for $N = 100$, 1000, 10000, 100000, 1000000.
- Fix $N = 1000000$ and $M = 100$. Report the run-times and speedups for $P = 1$, 2, 4, 8, 16, 32, 64, 128, 256.
- Fix $P = 32$ and $N = 100000$. Report the run-times and speedups for $M = 10$, 100, 1000, 10000.

Write some observations about the run-times and speedups that you recorded. In which case(s) does the data that you observe match your theoretical analysis from the previous question, and in which case(s) does it not match?

3. **Cross-Section Bandwidth** (27 points)

In the October 2 lecture, we talked about cross-section bandwidth and sorting. If you need a supplement to your notes, see the sketch at the end of this assignment. In this problem, we will extend this result to two- and three-dimensional tori.

(a) (5 points) What is the cross-section bandwidth of a P processor, two-dimensional torus? For simplicity, assume that P is a perfect square and that each link has unit bandwidth in each direction. The answer to this question is readily available by doing a suitable web search. That's OK. If you find an answer on the web or in a text book, you must cite your source. You must provide a short justification for your answer whether it's your own derivation or your paraphrase of a derivation that you cited.

(b) (5 points) Derive a lower bound for the time to sort N values using a two-dimensional torus consisting of P processors. You are expected to use an argument analagous to the one presented in class for sorting on a ring. Your answer should be a big-Ω statement, i.e. the time is $\Omega(P)$, or $\Omega(\log P)$ or something similar. Recall that $f(P) \in \Omega(P^2)$ means that f grows at least as fast as P^2 (to within constant factors). You should derive an informative bound, not a trivial bound such as $\Omega(1)$. See the notes at the end of this assignment if you need more hints.

(c) (5 points) Noting that sorting can be done in $O(N \log N)$ time on a sequential computer, the "ideal" time for sorting with P processors is $(N \log N)/P$. For what value of P does the communication time (from question 3b) match this ideal time? Write P as a simple expression of N.

(d) (5 points) What is the cross-section bandwidth of a P processor, three-dimensional torus? For simplicity, assume that P is a perfect cube and the rest of the guidelines from question 3a apply to this problem as well.

(e) (2 points) Derive a lower bound for the time to sort N values using a three-dimensional torus consisting of P processors. The guidelines from question 3b apply to this problem as well. This problem has fewer points than 3b because once you have solved that one, solving this question should be very easy.

(f) (3 points) For what value of P does the communication time (from question 3e) for sorting on a 3D-torus match the "ideal" time of $(N \log N)/P$? Write P as a simple expression of N.

(g) (2 points) Consider the case when $N = 2^{30}$ (roughly one billion). What are the values of P for which communication time matches the ideal time of $(N \log N)/P$ using your formulas from questions 3c and 3f?

Supplementary Material: Sorting on a ring: time bounds

For Question 3.

We showed a lower bound of $\Omega(N)$ time for sorting N values on a ring. We assumed that initially each of P processors held N/P items, and that after sorting, processor I must hold values $0..((I * N/P) - 1)$. The argument was based on allowing an adversary to chose the initial locations for the data values to make sorting difficult. To obtain a *lower* bound, we assumed a "magic" sorting algorithm that knows where every value should go and just sends it there. The key observation is that we can divide the ring into two halves with two bi-directional links connecting them. Let's call these the "left" and "right" halves of the ring. We then choose an initial distribution for values such that *every* value must traverse these links. If we assume that each link can convey one value per unit time, then $N/2$ values must cross the two links from left to right and likewise, $N/2$ values must cross the two links from right to left. This requires $(N/2)/2 = N/4 \in \Omega(N)$ time. We conclude that sorting takes $\Omega(N)$ time on a ring.

Now, consider speed-up for sorting on a ring. From the October 6 lecture, we have

$$SpeedUp = \frac{T(_{sequential})}{T(_{parallel})}$$

Comparison-based sorting requires sequential time of $\Theta(N \log N)$. If we want an "ideal" speed-up of P when using P processors, the lower bound for sorting on a ring indicates that $P \in o(\log N)$. This means that rings can achieve only very limited speed-up for sorting. For example, if N is one-billion (roughly 2^{30}), then a ring can achieve a speed-up of at most 30.