CpSc 418 **Homework 2** Due: February 7, 2023, 11:59pm
Early-Bird: February 5, 2023, 11:59pm

105 points

Prelude

Please submit your solution using:
`handin cs-418 hw2` Your solution should contain three files:

`hw2.erl`: Erlang source code for your solutions to the questions.

`hw2_tests.erl`: EUnit tests for the functions you were asked to write. In addition to the template that I will post, you can use `hw1_tests.erl` as an example.

`hw2.pdf`: Your solutions to written questions.

Templates for `hw2.erl` and `hw2_tests.erl` are available at
http://www.students.cs.ubc.ca/~cs-418/2022-1/hw/2/hw2.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.

Same notes as from HW1 about how your code should compile without errors or warnings, and your functions should only print stuff that is explicitly requested in the questions and you should acknowledge your collaborators and cite outside sources with a comment at the beginning of your `hw1.erl` file or a remark at the beginning of your `hw1.pdf` file. Thanks!

The Questions

1. π **reduce** (25 points): Recall the method of estimating π by choosing a positive integer, N, and finding the probability, $P(N)$, that two integers, `A` and `B` are co-prime. We then note $\lim_{N\to\infty} P(N) = 6/\pi^2$ and use this to compute π. For this problem, you may use the functions for trees of workers provided in wtree or from `red.erl`. I'll describe the questions using the wtree functions, but either approach is acceptable.

   (a) (15 points): Write a parallel implementation of the deterministic method for estimating π (see HW1 Q1.a&b). Write a function `p(W, N)` where `W` is a worker-tree returned by `wtree:create`, i.e. `wtree:create(23)` returns a binary tree of 23 worker processes. `p(W, N)` should return the number of pairs of integers $1 \leq$ `A`, `B` $\leq N$ such that `A` and `B` are co-prime.

   (b) (10 points): Measure the speed-up for your parallel implementation. Run your timing measurements on `thetis.students.cs.ubc.ca` and use time_it:t/1. `Thetis` has 64 cores (more technically, it has 32 hardware cores, each of which is 2-way multithreaded). Speed-up is T_{seq}/T_{par} where T_{seq} is the sequential running time (e.g. your solution from HW1), and T_{par} is the parallel run-time. Try experiments using 4, 16, 32, 64, 128, and 256 processes and values of `N` from 1000 increasing by multiples of 10 until you reach a value (for the number of processes) where the execution takes between 1 and 10 seconds. What number of processes and value for N gives you the highest speed-up?

   (c) **Just for fun:** Implement a parallel version of the randomized version from HW1 Q1.c and measure the speed-up.

2. **Reduce – by the book** (25 points): As noted in class, the course-library implementations of reduce and scan start with a master process broadcasting a request from the root of the process tree down to the leaves, the leaves do the work, and the results are combined back to the root and the grand total is sent to the master. When performing a scan, the course library adds a final top-down pass to generate the cumulative sums (or whatever operator was used) at the leaves. In the book, reduce and scan are both start with a bottom-up pass that is initiated by the leaf processes, followed by a top-down pass the distributes the results. Scan uses the same bottom-up pass as reduce, but the final top-down pass is different for scan and reduce, both by the books method and the course library.

   For this question, implement

```
reduce_bu(ProcState, LeafVal, Combine) -> % Total
  % ProcState: the process state for this worker.
  % LeafVal: the value at the leaf of the reduce tree computed by this worker.
  % Combine: the associative function for combining values.
```

   to work with worker trees as constructed by `red.erl:create/1`. The function `reducc_bu/3` is called by each leaf of the worker tree to compute the parallel reduce. Note: `ProcState` is needed by `reduce_bu` because it includes entries for:

   `parent`: The pid of the parent node for this process, or the atom `none` if this is the root node.

   `children_bu`: A list of pids of the children of this tree, in bottom-up order. To describe this list, I'll assume that that `P`, the number of worker processes in the tree is a power of 2. If the index of a worker node is `I` we can write $I = 2^K J$ where either J is odd, or $J = 0$ and $K = log_2 P$. Node `I` has K children in the list associate with `children_bu` – these are the workers with indices $I + 2^M$ for each $M \in \{0, 1, \ldots, K-1\}$. The list is in order of increasing worker index. For example, if `P` = 16, then $\texttt{worker}_0$ has the list of children:

   $[\texttt{worker}_1, \texttt{worker}_2, \texttt{worker}_4, \texttt{worker}_8]$

   and $\texttt{worker}_{12}$ has the list of children

   $[\texttt{worker}_{13}, \texttt{worker}_{14}]$

   If `P` is not a power of two, we construct a binary tree where the left subtree has `ceil(P / 2)` leaves, and the right subtree has `floor(P / 2)` leaves, and likewise for the subtrees. If a (sub)tree spans workers `W_I` through `W_J`, then worker `W_I` is the root of the (sub)tree and its left subtree. There are `M = J + 1 - I` leaves to this tree, and worker `W_{I + (M div 2)}` is the root of the right subtree.

   Hint: Here's a sketch of `reduce_bu`:

   - Fetch the list of child pids and the parent pid from `ProcState`.
   - Call a recursive helper function that walks down the list of child pids, receiving value from each, and combining them. Make sure that you keep the left-right order of arguments correct. I'll test your `reduce_bu` with a `Combine` function that is associative but not commutative.
   - When you reach the end of the list, send your result to your parent (if you have one).
     - If your parent is `none`, then propagate the result back down the tree.

   Each worker process calls `reduce_bu` when its `LeafVal` is ready. The values are `Combine`'s up the tree, and the final total propagated down the tree to each worker process.

3. **Scan – by the book** (20 points): For this question, implement

```
scan_bu(ProcState, AccIn, LeafVal, Combine) -> % AccLeft
  % Parameters:
    % ProcState: the process state for this worker.
    % AccIn: the initial value for the accumulator (used at the root of the tree).
    % LeafVal: the value at the leaf of the reduce tree computed by this worker.
    % Combine(Left, Right): the associative function for combining values.
  % Result:
    % AccLeft: % the result of combining everything to the left of this node using Combine.
               % If this is the leftmost node, then AccLeft =:= AccIn.
```

Each worker process calls `scan_bu` when its `LeafVal` is ready. The values are `Combine`'d up the tree. When the `Combine` operation reaches the root of the tree, you need to propagate values back down the tree. This is where the code for `scan` differs from that for `reduce`. Many of the ideas from your solution to `reduce` should carry over to `scan`.

The return value from `scan_bu` is the combined value of everything to the left of this node starting with the value `AccIn`. The worker process can combine this with its local value to get its segment of the scan. See `scan_x(P, EUnitFlag)` in `hw2_tests.erl` for an example of using `hw2:scan` to compute the cumulative sum of the elements of a list distributed across a tree of `P` workers. The `EUnitFlag` determines whether `scan_x` returns a list of EUnit tests or if it runs the tests and prints the result in a (hopefully) human-friendly way.

4. **Brownian motion** (simplified, 20 points): Consider a random walk that starts at the origin of the (X, Y) plane. In Erlang, we will represent coordinates as tuples `{X, Y}` (although lists might have been a better choice to generalize to other dimensions). At each step, the walker chooses a random direction (uniformly in $[0, 2\pi)$ where the direction is the angle, in radians, counter-clockwise from the positive X-axis. The walker also chooses how far to go in this step, R, where R is exponentially distributed with parameter `Alpha`. We are interested in what is the greatest distance from the origin that our walker reaches during a `N` step walk. The template code in `hw2.erl` provides a sequential implementation.

   (a) (20 points): Write a parallel implementation: `brownie_par(W, N, Alpha) -> GreatestDistance`. Use the functions from the `wtree` module in the class library. Hint: my solution uses:

   `wtree:nworkers(W)`: – find out how many leaf processes there are to know how much work to give to each worker.

   `wtree:update(...)`: – each worker computes its sequence of steps.

   `wtree:scan(...)`: – Each worker computes a list of the locations reached after performing each of its steps. Note: the "walk" by worker I (for `I > 0`) starts from the end point of the walk by worker `I-1`.

   `wtree:reduce(...)`: – find the maximum distance from the origin.

   (b) **Optional/extra-credit**: This problem has been rendered unreasonable for students (but just fine for the instructor) because new release of Erlang is a memory hog and student accounts have small per-process memory limits. There will be 5 (or maybe a bit more) points of extra-credit for reporting any reasonable results.
   Compute the speed-up for various numbers of worker processes and various choices of `N`. You can fix `Alpha = 1.0`. Follow the guidelines for measuring speed-up that were given in Q1.

   (c) Just for fun: the estimate of max-distance has a *large* variance. I'm observing a variance of about 1/3 the mean. Don't let this stress you out. If you want more consistent results, try computing the average from 100 to 1000 runs. Note that this provides another opportunity for parallelism.

5. **Speed-up** (15 points): Consider using reduce on a problem of size N using P processors. For simplicity, assume that P is a power of two – thus, the reduce tree is a complete binary tree. Each leaf worker takes

time $(N/P) * T_{leaf}$ time to compute it's local result. The time to send a message between processes is λ, and the time to combine the results of two sub-trees (not including the communication time) is $T_{combine}$.

(a) (easy, 3 points) What is the total time to compute the reduce?

(b) (easy, 2 points) What is the total time to compute the same result sequentially? Remember: no communication costs.

(c) (5 points) If $T_{combine} = T_{leaf}$, how big does N need to be to get a speed-up of $P/2$? Assume that $\lambda \gg T_{leaf}$ (had been $\lambda \gg 1$ but T_{leaf} is what matters) and simplify accordingly. Your solution should make it clear where you use this assumption.
Note: several students have noted that you can derive an exact formula for N without making a the simplifying approximation that comes by assuming that λ is large. Indeed, you can. The exact formula has a sum in it. If you assume λ is large, then you can eliminate one term from the sum to get a formula where it's obvious how N depends on P and λ. In other words, if you're working on a project in real-life and asked "If we increase the number of processors by 10×, can we make this go at least 1.8 times faster?" you can quickly estimate it. Often, system design needs an ability to quickly eliminate unworkable solutions without having to do detailed calculations first. That's why simplifying to the asymptotic form is really good for getting intuition about the trade-offs.

(d) (5 points) If $T_{leaf} = \lambda$, how big does N need to be to get a speed-up of $P/2$?