CpSc 418 **Homework 2** Due: Oct. 9, 2012, 11:59pm

Revised September 20, 2012

65 points.
5% extra credit for problems 1–3 if solution submitted by 11:59pm on Oct. 4.

Please submit your solution using the handin program. Submit the program as
cs418 hw2
This requires you to have an account on the UBC Computer Science undergraduate machines. If you need an account, go to: `https://www.cs.ubc.ca/students/undergrad/services/account`
to request one.
Your submission should consist of the following files:

`hw2.erl` – Erlang source (ASCII text). All functions requested in this assignment must be exported by this module.

`hw2.txt` – plain, ASCII text.

`hw2_data.txt` – plain, ASCII text.

`hw2_plotN.[jpg|gif|pdf|svg]` – image files of plots, where `hw2_plotN` is replaced with `hw2_plot1`, `hw2_plot2`, etc.

The first file, `hw2.erl`, will be your solution to the programming part of the assignment, neatly commented.
The second file, `hw2.txt`, will be a plain text file with your solution to the written part of the assignment.
The third file, `hw2_data.txt`, will be a plain text file with your raw data for the written questions, clearly numbered by question.
The fourth-Nth files, `hw2_plotN.[jpg|gif|svg]`, will be image files in jpg, gif, or svg format of any plots used in arriving at the solution. Note that it is acceptable for your submission not to include these files.

Note that no other file formats will be accepted. Your answers to the written questions must be **short and concise**, but complete. Overly long and wordy answers may be docked points. When presenting answers to written questions, put the numerical answers each on a line, followed by a blank line, followed by any explanations, discussions, and justifications.

1. **Head vs. Tail Recursion (20 points)**
   This problem is an introduction to measuring execution time using the `time_it` module. It also provides a comparison of a head-recursive and tail recursive implementations of a function.

   (a) **Head-recursive implemenation of `max` (5 points)**
   Write a head-recursive function

   ```
   max_hr(List) -> Number
   ```

   that takes as an argument a list of numbers (i.e. integers or floats) and returns the largest element of the list.

   (b) **Tail-recursive implemenation of `max` (5 points)**
   Write a tail-recursive function

   ```
   max_tr(List) -> Number
   ```

   that takes as an argument a list of numbers (i.e. integers or floats) and returns the largest element of the list.

   (c) **Performance comparsison (10 points)**

   ```
   t(Fun) -> [mean,Mean,std,Std]
   ```

in the `time_it` module is a function that computes the mean and standard deviation of the time spent when evaluating function `Fun`. It calls `Fun` until a total execution time of 1 second is reached or exceeded. Use the `time_it:t` function to compare the time for evaluating `max_hr` and `max_tr` on lists of length 10, 100, 1,000, 10,000, 100,000, 1,000,000, and 10,000,000. Include the raw data in `hw2_data.txt`. In `hw2.txt`, answer the following questions:

i. Which is faster, `max_hr` or `max_tr`?

ii. When is the difference the most significant?

iii. Why does the relative difference depend on the length of the list?

2. **Message Performance (30 points)**

   How fast can Erlang send messages between two processes?

   In this problem, we consider a linear model of the time it takes two processes to do a variable amount of work, and send a small, fixed, constant-size message to each other. A simple model for such an interaction could be:

$$T(n) = t_0 + t_1 * W(n)$$

   where $T(n)$ is the total time taken, t_0 is the time it takes to pass a small, fixed message without either process doing any work, $W(n)$ is a work function that varies the amount of work it does according to the (positive) integer variable n, which specifies the number of computations $W(n)$ will do, and t_1 is the slope of this function (that is, the change in elapsed time that it takes to perform work and message passing as n increases).

   However, this model is flawed when Erlang runs locally and computation $W(n)$. When $W(n)$ is large enough, Erlang will create a separate thread for each process, but when $W(n)$ is small, Erlang will not run separate threads, instead treating the two processes as coroutines. Instead, the model will be piecewise linear with two pieces:

$$T(n) = \begin{cases} t_0 + t_1 W(n), & \text{if } n \leq n_0 \\ u_0 + u_1 W(n), & \text{otherwise} \end{cases}$$

   In this problem, you will write a program and take measurements to determine some model parameters.

   (a) **Multithreading (10 points).** Find and report the amount of work at which Erlang begins to run two threads (i.e. find n_0)

   (b) **Message Overhead (10 points).** Find and report the amount of time needed for Erlang to send a message without doing any work (i.e. find t_0)

   (c) **Message Overhead (10 points).** Find and report the change in running time as n scales up for the multiple-threads case, and for the coroutine case (i.e. find t_1 and u_1)

   For the above problems, include a presentation of your raw data and any plots used to determine the values.

3. **Parallelism Primes (35 points).**

(a) **Parallel Primes (20 points).** Here's a sequential implementation of the sieve of Eratosthenes:

```
% primes(A, N) -> the listof all primes P with A ≤ P ≤ N.
%      The list is in ascending order.
primes(Lo, Hi) when is_integer(Lo) and is_integer(Hi) and (Lo > Hi) -> [];
primes(Lo, Hi) when is_integer(Lo) and is_integer(Hi) and (Hi < 5) ->
   lists:filter(fun(E) -> (Lo =< E) and (E =< Hi) end, [2,3]);
primes(Lo, Hi) when is_integer(Lo) and is_integer(Hi) and (Lo =< Hi) ->
   M = trunc(math:sqrt(Hi)),
   SmallPrimes = primes(2, M),
   BigPrimes = do_primes(SmallPrimes, max(Lo, M+1), Hi),
   if
      (Lo =< 2) -> SmallPrimes ++ BigPrimes;
      (Lo =< M) -> lists:filter(fun(E) -> E >= Lo end, SmallPrimes)
                     ++ BigPrimes;
      true -> BigPrimes
   end.
primes(N) -> primes(1,N). % a simple default

% do_primes(SmallPrimes, Lo, Hi) the elements of [Lo, ..., Hi] that are not divisible
%      by any element of SmallPrimes.
do_primes(SmallPrimes, Lo, Hi) ->
   lists:foldl(fun(P, L) -> lists:filter(fun(E) -> (E rem P) /= 0 end, L) end,
               lists:seq(Lo, Hi),
               SmallPrimes).
```

where L is a list of all primes between the parameters A and N, that is all primes p such that $A \leq p \leq N$.
Implement a function `par_primes(A, N, Nproc) -> L`
that use Nproc processes to find the primes from A to N. Use worker pools as provided in the `workers` module – see `http://www.ugrad.cs.ubc.ca/~cs418/2012-1/src/erl/source.html` In `hw2.txt` give a brief explanation of how you exploited parallelism for this problem including the reasons behind your most important design choices.

(b) **Speedup (5 points).** Using the functions in the `time_it` module available on the course page – see `http://www.ugrad.cs.ubc.ca/~cs418/2012-1/src/erl/source.html`, measure the speedup that results in evaluating `par_primes(1000, 500000, 64)` function from part 3a over the sequential `primes(1000, 500000)` given when running the two functions on `gambier`. Report this number to the nearest integer (or the nearest 0.5 if your speedup factor is below 5), and also include your raw data (the two numbers used to calculate your speedup). Is this the speedup you would expect? Why is this (or is not) the case?

(c) **How Parallel? (10 points).** Using the `time_it` module, measure the running time of your `par_primes` function (when running on `gambier`) for different numbers of Erlang processes used in the worker pool. Specifically, measure how long it takes `par_primes(1000,100000,Nproc)` to run when `Nproc` $= 1, 2, 3, 4, \ldots, 256$. You don't need to run it for all 256 values of `Nproc`, but you should try every value of `Nproc` from 1 to 16, and enough from 16 to 256 to show a clear pattern. You should show that run-time initially decreases with increasing `Nproc` and then reaches a minimum. After that, using any more Erlang processes will actually slow the problem down (think about why this is the case). If you look at the data carefully, you'll probably observe some zig-zagging of the plots. Getting a reasonable approximation of the minimum is good enough for this problem.

Your task will be to report for which value of `Nproc` the running time is minimized. Is this value of `Nproc` what you expected? If not, what is the value that you expected, and is the actual value you observed reasonable? Why might there be discrepancies between the expected and observed values? Also include your raw data and a plot.