Exam in Parallel Functional Programming

08:30–12:30, Tuesday, May 22nd, 2012..
Lecturers:
John Hughes, tel 1001
Mary Sheeran, tel 1013

Permitted aids:
Up to two pages (on one A4 sheet of paper) of pre-written notes. These notes may be typed or hand-written. This summary sheet must be handed in with the exam.

There are X questions. 24 points is required to pass (grade 3), 36 points is required for grade 4, and 48 points is required for grade 5.

1. **Parallel Functional Programming** **10 points**

   (a) Why are functional languages particularly well-suited to parallel programming? **1 points**

   (b) An easy way to parallelize functional programs is to evaluate every expression in parallel. Would you recommend this approach? Explain your answer briefly. **1 points**

   (c) "After parallelization, any program should be able to run N times faster on N cores." Is this true or false? Explain your answer briefly (for example, with reference to *Amdahl's Law*). **1 points**

   (d) What is the connection between associative operators and parallelism? **1 points**

   (e) Do parallel processes share memory in

      i. Haskell? **1 points**

      ii. Erlang? **1 points**

   (f) What is the main advantage of the *strategies* approach to parallelism in Haskell? **1 points**

   (g) What is the effect of linking two Erlang processes? **1 points**

   (h) Erlang is used to build robust systems. How can this be reconciled with the Erlang slogan "*Let it crash*"? **1 points**

   (i) What is the purpose of a supervisor? **1 points**

2. **Parallel Sorting** **8 points**

   (a) Read this Haskell definition of merge sort:

```
merge_sort [] = []
merge_sort [x] = [x]
merge_sort xs = merge (merge_sort ys) (merge_sort zs)
  where (ys,zs) = split xs

merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) | x <= y    = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys

split xs = split' [] xs xs

split' xs (y:ys) (_:_:zs) = split' (y:xs) ys zs
split' xs ys _            = (xs,ys)
```

   Using `par` and `pseq`, write a parallel version of `merge_sort`. Ensure that the task granularity is not so fine that the overheads of parallelism dominate the run time. You may reuse the functions defined above without including their definitions in your answer. **4 points**

(b) Read this Erlang definition of merge sort (which is simply a translation of the Haskell version):

```
merge_sort([]) -> [];
merge_sort([X]) -> [X];
merge_sort(Xs) ->
    {Ys,Zs} = split(Xs),
    merge(merge_sort(Ys),merge_sort(Zs)).

merge([],Ys) -> Ys;
merge(Xs,[]) -> Xs;
merge([X|Xs],[Y|Ys]) when X =< Y ->
    [X|merge(Xs,[Y|Ys])];
merge([X|Xs],[Y|Ys]) -> [Y|merge([X|Xs],Ys)].

split(Xs) -> split([],Xs,Xs).

split(L,[X|R],[_,_|Xs]) -> split([X|L],R,Xs);
split(L,R,_)            -> {L,R}.
```

Using `spawn_link`, `self`, and message passing, write a parallel Erlang version of `merge_sort`. As above, ensure that the task granularity is not so fine that the overheads of parallelism dominate the run-time. Once again, you may reuse functions defined above in your answer without including their definitions. **4 points**

3. **The Par-monad** **8 points**

(a) Using the Par-monad, define a parallel fold function with the type

```
parFoldM :: NFData a => (a -> a -> Par a) -> [a] -> Par a
```

You need not take account of task granularity or thresholding. **2 points**

(b) Write a parallel divide-and-conquer higher-order function in Haskell for use in the Par-monad. If you wish, you may use

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
```

4 points

(c) Redefine merge sort using your divide-and-conquer function. **2 points**

4. **Work and Depth** **8 points**

(a) Briefly explain, using at least one small example program, the notions of work and depth (or span) as presented by Blelloch. How does expected running time relate to work, depth and number of processors? State one aspect of the cost of running parallel computations which the basic work / depth model does *not* cover? **3 points**

(b) The following is Blelloch's pre-scan function (for inputs whose length is a power of two):

```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});

scan_op(max, 0, [2, 8, 3, -4, 1, 9, -2, 7]);

it = [0, 2, 8, 8, 8, 8, 9, 9] : [int]
```

What are the work and depth for this form of pre-scan, in terms of n, the length of the input sequence? **2 points**

(c) Consider the following variant of the stock market problem: given the price of a stock at each day for n days, determine the biggest profit you can make by buying one day and selling on a later day. A simple sequential (serial) solution requires O(n) work for an input sequence of length n. In NESL, the problem can be solved as follows:

```
function stock(a) =
  max_val({x - y : x in a; y in min_scan(a)});
```

It uses `min_scan` (with the same work and depth as `scan_op` above, and without the restriction on the input length) and `max_val`, which is a parallel fold. Is the work of this parallel solution still $O(n)$? Explain your answer. What is the depth of the solution? **2 points**

(d) In Haskell, one can solve the above stock market problem using a single parallel fold. What is the work and depth in that case? **1 points**

5. **Parallel Map** **12 points**

(a) The following Erlang function is intended to deliver the same result as `lists:map(F,Xs)`, but to compute the elements of the resulting list in parallel.

```
pmap(F,Xs) ->
    Parent = self(),
    [begin
         spawn_link(fun() -> Parent ! F(lists:nth(I,Xs)) end),
         receive FX -> FX end
     end
     || I <- lists:seq(1,length(Xs))].
```

The following functions are used to test it:

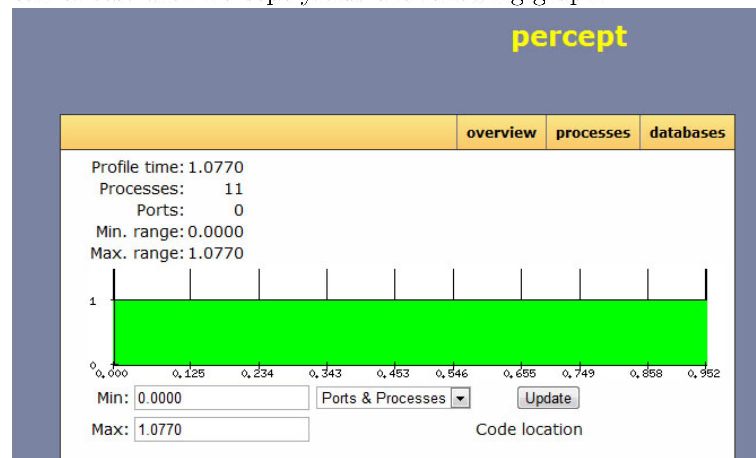
```
expensive(X) ->
    _ = length(lists:seq(1,1000000)),
    X+1.

test() ->
    pmap(fun expensive/1,lists:seq(1,10)).
```

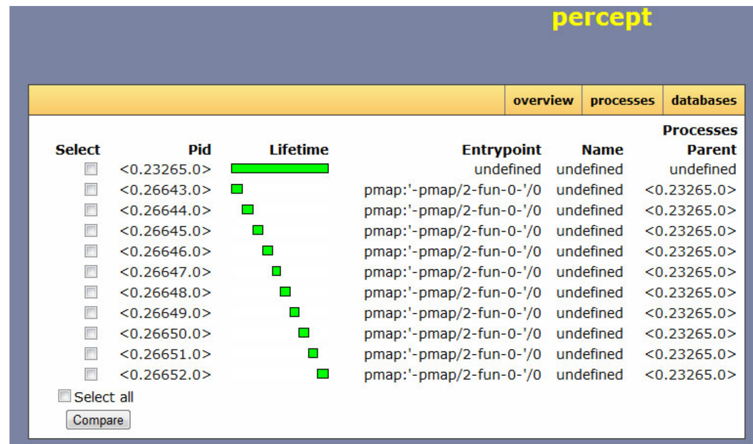
Timing calls of these functions in the Erlang shell on a dual core machine yields:

```
1> timer:tc(pmap,expensive,[0]).
{125000,1}
2> timer:tc(pmap,test,[]).
{1077000,[2,3,4,5,6,7,8,9,10,11]}
```

(with a timer resolution of about 15000 microseconds). Profiling a call of test with Percept yields the following graph:



and examining the process lifetimes yields:



i. How many processing cores can this version of `pmap` make good use of? **1 points**

ii. Why can't this code use 10 cores in parallel in this example? **1 points**

iii. All the elements of the resulting list are sent back to the same `Parent` process. How can we ensure they appear in the correct *order* in the final result? **2 points**

iv. The code above incurs unnecessary communication costs. Explain why. **1 points**

v. Write a corrected version of `pmap` that addresses all of these issues. **3 points**

(b) Recall that `spawn_link(Node,Fun)` spawns a process that calls `Fun()` on the Erlang node `Node`. Write a function `dmap(Fun,Xs)` that implements a *distributed* map function, evaluating calls of `Fun` on all the connected nodes. Each node should be given *one* call of `Fun` to evaluate at a time; while list elements remain to be processed, then each node should be given another element to work on as soon as its previous task is done. Make sure you return the list of results in the correct order. **4 points**

Hint: If `XYs` is a list of pairs with different first components, then `lists:sort(XYs)` will sort the list by the first components (because Erlang's ordering on pairs is lexicographic: the first components are compared, and only if they are equal are the second components compared). For example,

```
1> lists:sort([{2,a},{3,b},{1,c}]).
[{1,c},{2,a},{3,b}]
```

6. **Map-Reduce** **6 points**

   (a) `map_reduce` takes a mapper function, a reducer function, and input data as parameters. Consider a naïve version in which the input data is represented as a list. If `map_reduce` were defined in Haskell, what would its type be? You need not include any class constraints, such as `Eq a`, in the type that you give. **1 points**

   (b) Suppose the input data to `map_reduce` consists of pairs of a page number and a list of words, such as

   ```
   [{1,["hello","clouds"]},{2,["hello","sky"]}]
   ```

   Write a mapper and a reducer function to convert this to an index of words and page numbers... in this example,

   ```
   [{"clouds",[1]},{"hello",[1,2]},{"sky",[2]}]
   ```

   2 points

   (c) Given the same input data, write a mapper and a reducer function to associate each word with its total number of occurrences. Recall that a word may occur several times on the same page. **2 points**

   (d) In a distributed implementation of Map-Reduce, why might we wish to use the reducer function in the map jobs as well as the reduce jobs? **1 points**

7. **Choosing between approaches to parallel functional programming** **8 points**

   Consider the situation (described by Lennart Augustsson in his guest lecture) where you have embarassingly parallel computations and a remote grid machine that does stateless computation tasks. (Note: In questions 7a and 7b below, there is no one correct answer. The important thing is to be able to justify your choice based on results that you obtained using the different approaches in the labs (and possibly also on what you have heard from guest lecturers). Imagine pitching your chosen approaches to a sceptical manager.)

   (a) Would you choose Erlang or Haskell in this setting? Briefly explain your choice. **3 points**

   (b) If using Haskell, and given a choice between using the Repa library, Strategies and the Par monad, which would you choose? Again, explain *briefly* why you make this choice. **5 points**