Exam in Parallel Functional Programming

08:30–12:30, Tuesday, May 28th, 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 7 questions.
24 points are required to pass (grade 3), 36 points are required for grade 4, and 48 points 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) Briefly explain the `par` and `pseq` approach to expressing parallelism in Haskell **1 points**

   (e) Briefly explain the use of Divide and Conquer in parallel programming, for example by writing and explaining a `divConq` combinator for use with the *Par monad*. **2 points**

   (f) Name one advantage of the *Par Monad* 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. **Obsidian** **10 points**

   (a) Identify some functions (for example `force`) in Obsidian that give the user control over important aspects of the GPU. Explain those functions briefly. **2 points**

   (b) Obsidian makes use of Push and Pull arrays. Outline their main differences. **2 points**

   (c) This is the definition of a Pull array

   ```
     data Pull s a = Pull s (EWord32 -> a)
   ```

   Implement `fmap`, which maps a function onto each element of the array:

   ```
   fmap f (Pull s ixf) = Pull s $ ???
   ```

   1 points

   (d) This is the definition of a Push array

   ```
     data Push t s a =
           Push s ((a -> EWord32 -> Program Thread ()) -> Program t ())
   ```

   Now implement `fmap` for Push arrays:

   ```
   fmap f (Push s p) = Push s $ ???
   ```

   2 points

   (e) Implement a parallel reduction (with the functionality of `foldl1`) in Obsidian so that a reduction kernel in CUDA could be generated. Explain your implementation. **3 points**

4. **Work and Depth** **7 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? **2 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)});
```

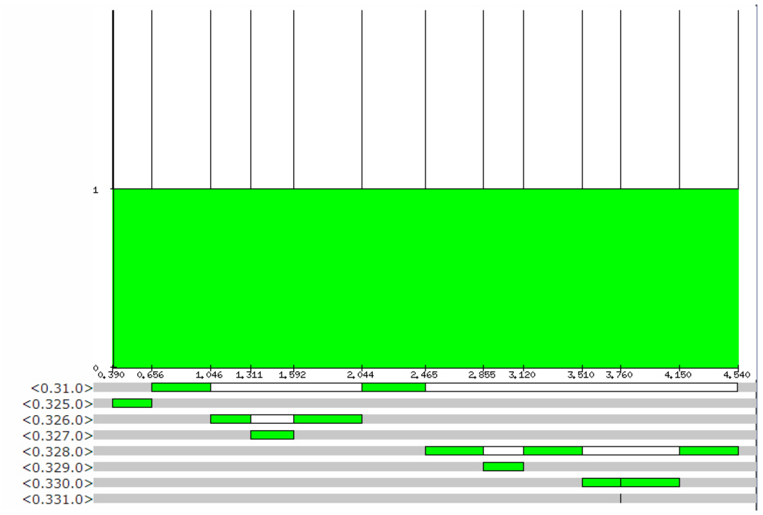
It uses `min_scan` (with the same work and depth as `scan_op` above, and without the restriction on the input length) and `max_val`, which is a parallel fold. Is the work of this parallel solution still $O(n)$? Explain your answer. What is the depth of the solution? **2 points**

(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 Reduce** Read the following definition of a parallel reduce function. **12 points**

```
reduce(_,[X]) ->
    X;
reduce(F,[X,Y]) ->
    F(X,Y);
reduce(F,L) ->
    {L1,L2} = lists:split(length(L) div 2,L),
    Parent = self(),
    Y = reduce(F,L2),
    Pid = spawn_link(fun() -> Parent ! {self(),reduce(F,L1)} end),
    receive
{Pid,X} ->
    F(X,Y)
    end.
```

Profile a call of `reduce` with Percept on a list of 15 elements resulted in the following graph:

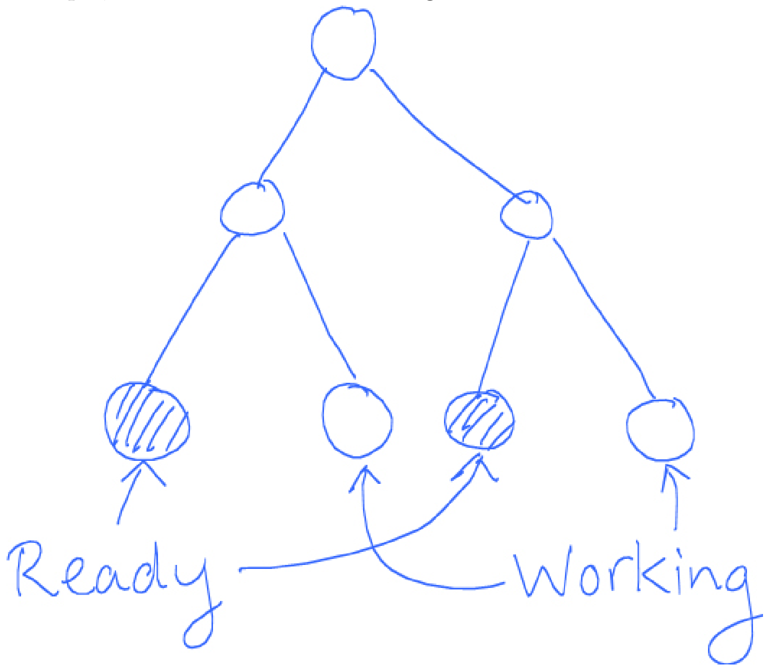


(a) How many processing cores can this version of `reduce` make good use of? **1 points**

(b) Why can't this program take advantage of a larger number of cores? **1 points**

(c) Make a small fix to the code above so that it makes better use of parallelism. You need not copy the entire definition into your answer: just write the modified lines and explain clearly where in the code they should be placed. **1 points**

(d) A weakness of the code above (even after your fix) is that, if recursive calls to `reduce` take widely differing amounts of time, then cores may remain idle even though there is work that could be done. For example, in the situation in this diagram



then two recursive calls are ready, while two more are currently being evaluated. The code above will wait for each busy call to terminate before combining its result with its neighbour; however, a smarter implementation could begin to combine the two available results already to use more parallelism. (Of course, this changes the *order* in which results are combined, but we will assume this does not affect the final result). Write a new version of `reduce` which uses this idea to combine the results of recursive calls as soon as any two are available. **4 points**

(e) Recall that `spawn_link(Node,Fun)` spawns a process that calls `Fun()` on the Erlang node `Node`. In a distributed system, we might want to run reduce jobs on different nodes in the network to share the workload. Write a *load balancing* server, which accepts requests of the form `{call,Pid,F}`, calls `F()` on one of the nodes of the network, and then sends the result back to `Pid` in a message of the form `{result,Res}` (where `Res` is the value that `F()` returned). You should ensure that you can make use of all nodes in your network, but that each node has at most one job to execute at a time. **4 points**

(f) What modification would you make to your `reduce` function to make use of the load balancer you wrote in question 5e? **1 points**

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**

   (a) Imagine that you work at a company that builds applications that are highly data parallel. The possibility of using Haskell is being discussed. Your manager asks you to write a brief description of the Repa library for parallel array processing in Haskell. Write that description. It should include some example code, and a discussion of the pros and cons of using Repa, based on your own experience of using it. **4 points**

   (b) Explain how fault tolerance is handled in Erlang, using concrete code snippets. (You might, for example, explain how the Map-Reduce example from above could be made fault tolerant.) **4 points**