CHALMERS TEKNISKA HÖGSKOLA
Dept. of Computer Science and Engineering

14:00–18:00, Thursday, Augest 20, 2020.
Parallel Functional Programming
DAT280, DIT261

Exam in Parallel Functional Programming

14:00–18:00, Thursday, Augest 20, 2020.
Examiners / Teachers :
John Hughes (rjmh@chalmers.se)
Mary Sheeran (mary.sheeran@chalmers.se)

Attempt all questions. 24 points are required to pass (grade 3), 36 points for grade 4, and 48 points for grade 5.
Our expectations. We expect you to write, run and benchmark programs in Haskell and Erlang during the exam. Success in this will increase your chances of passing. However, it is important not to get stuck in small technical details (such as syntax), and to think carefully about how you use your time during the exam.
Questions for the examiners or teachers Write "Question for examiner" in the zoom chat. You will be transferred to the examiner's breakout room in due course.
Permitted Aids. All aids are permitted. If you include code or text from any source (a book, paper, slides, web page etc.), cite your source clearly. Communication with others, including digital communication, is **strictly forbidden**.
Please be specific and concise when answering questions. Short, polished text beats rambling, generic text. Keep to the point of the question! Nonsense will damage the overall impression, even if other parts of your answer are reasonable.
What to hand in. For each question, submit one or more files containing your answer. Each file should be a .hs or .erl file containing code and comments, a pdf containing text and diagrams (possibly scanned), or a plain text file if needed. Zip together all of your files and submit at the end of the exam. Mark question numbers clearly!
How to hand in your solutions. You should submit via Canvas if possible. If, for some reason, Canvas is not available at the end of the exam, submit your solution via Fire (as for labs). If that does not work either, submit by email to mary.sheeran@chalmers.se, with the exact title *Exam Submission TDA280* for Chalmers students and *Exam Submission DIT261* for GU students. It is your responsibility to keep within the allotted time.

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

(a) In what way does shared mutable data cause a problem in parallel programming? **1 point**

(b) If offered a choice between using `par` and `pseq` or the `Par` monad to parallelise Haskell programs, which would you choose? Explain why. **1 point**

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

(d) "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 point**

(e) Pick a small Futhark function from your Lab C solution. Include both the code and your explanation of what it does and what the types mean. **1 point**

(f) After an Erlang process sends a message, using

```
Pid ! Msg,
```

does the sending process continue its execution

i. immediately,
ii. once the message is safely delivered to the recipient's mailbox,
iii. once the recipient has `received` the message from its mailbox,
iv. or once the recipient has sent a reply?

1 point

(g) What is a network partition? **1 point**

(h) Explain the CAP theorem: what do C, A and P refer to, and what does the theorem state? **1 point**

(i) Suppose three V-nodes in a Dynamo-style eventually consistent database hold three different values for the same key, with the vector clocks shown:

Value	Vector clock
1	`[{a,1},{b,2}]`
2	`[{b,1},{c,1}]`
3	`[{a,2},{b,2},{c,1}]`

i. Which values, if any, are in conflict? **1 point**

ii. After the database reaches an eventually consistent state, what value will the key be associated with? **1 point**

2. **Work and Depth (and control of granularity)** **24 points**

Answer in files called **trees.hs** for code, and **trees.pdf** for the rest.

(a) Why is it that a `map` of a function over a list in Haskell is well suited to parallelisation? What are the work and depth (or span) of `map f`, assuming work and depth 1 for `f`? **2 points**

(b) Your next task will be to parallelise a Haskell `map`. You are given a tree data structure and functions to make completely balanced binary trees of a particular depth, and to construct lists of such trees:

```
data Tree = Nil | Node !Int !Tree !Tree
  deriving Show

make :: Int -> Int -> Tree
make i 0 = Node i Nil Nil
make i d = Node i (make (i2-1) d2) (make i2 d2)
  where i2 = 2*i; d2 = d-1

makes :: Int -> Int -> [Tree]
makes n k = [make i k | i <- [1..n]]
```

Now, you are asked to apply the function `check` to each element of such a list of trees (and fully evalute the result). Make the list length somehow related to the last 4 digits of your personal number (and make up 4 digits if necessary). So, for instance, I used `makes 22660 8` in benchmarking, as my last 4 digits are 2266.

```
check :: Tree -> Int
check Nil          = 0
check (Node i l r) = i + check l - check r
```

(Hint: The exclamation marks in the definition of `Tree` indicate strict evaluation of sub-parts of the tree, and help us to avoid problems with memory use and garbage collection. You should include them. (The code comes from benchmarks of GHC's memory management.) You will likely need to use GHC flags to control garbage collection. I used `-A750M` to control the nursery size, see notes for Lab A in the course.)

First, time how long it takes to `map` the `check` function over your list of trees without using parallelism. **1 point**

(Hint: as a reminder of how criterion (in the version I have installed) works, this code times both `check` and `map check` running sequentially:

```
import Criterion.Main
...
main = benchmarkSeq

benchmarkSeq :: IO ()
benchmarkSeq
  = do
     let t = make 1 24
         ts = makes 22660 8
     defaultMain [bgroup "reduceQSeq"
                  [bench "seqCheck"      (nf check t),
                   bench "seqChecks"     (nf (map check) ts)
                  ]]
```

You should only use the `-threaded` flag to GHC for parallel code.)

(c) Next, use `parMap` from the Par monad to make use of parallelism. (You do not need to define `parMap` as it is available in `Control.Monad.Par`) Benchmark the result. Explain why this does not work well. **3 points**

(d) Whatever you answered, modify your function to give better control of task granularity, for example by using chunking. Benchmark and comment upon the result. **4 points**

(e) Next, we turn our attention to the `check` function itself. Use the Par monad to parallelise it, first without control of granularity. Call the resulting function `pCheck`. Benchmark and comment upon the result. **4 points**

(f) Next, add a parameter to control task granularity. Benchmark the result for a number of different choices of that parameter. Call the function `pCheckG`. Comment on the success (or failure) of your new parallelisation. **3 points**

(g) Estimate the work and depth (or span) of the sequential `check`, and of `pCheck` and `(pCheckG k)` for a tree of depth d, assuming that work 1 is done at each node of the tree (and ignoring the overheads of parallelism). **3 points**

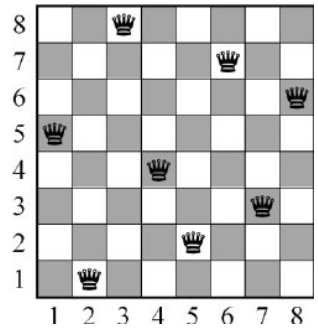
(h) Blelloch's work on NESL placed great emphasis on the importance of parallel prefix sum, or *parallel scan*. What are the work and depth (or span) for this parallel (pre)scan in NESL? Explain your answer. **2 points**

```
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});
```

(i) Explain in your own words how Brent's theorem relates the time to run an algorithm on n processors to the work and depth studied by Blelloch. **2 points**

3. **Parallel Erlang Programming** **20 points**

The **eight queens problem** is a well-known puzzle, involving placing eight queens on a chessboard so that no queen threatens another—that is, so that no queen is on the same row, in the same column, or on the same diagonal as another. One solution to the puzzle is shown below:



Since every column must contain exactly one queen, a solution can be represented conveniently as a list of eight row numbers—for the example above, as `[5,1,8,4,2,7,3,6]`.

There are many different ways to solve the puzzle—in fact, there are 92 different solutions. In this question we are going to develop parallel code to compute the list of *all* solutions to the puzzle. Because the puzzle can be solved very fast on an 8x8 chessboard, we will consider generalization to the N-queens problem, of placing N queens on an $N \times N$ chessboard under the same conditions, and we will solve the 12×12 case as a benchmark.

The (sequential) code below can be used to find all the solutions to the N-queens problem.

```
-module(queens).
-compile([export_all,nowarn_export_all]).

queens(N) ->
  queens(N,N).

%% queens(M,N) returns all ways to place M queens on an MxN chessboard.
queens(0,_N) ->
  [[]];
queens(M,N) ->
  [[Q|Qs] || Qs <- queens(M-1,N),
             Q  <- lists:seq(1,N),
             safe(Q,Qs)].

%% safe(Q,Qs) is true if a queen can safely be placed on row Q, given
%% that the columns to the right contain queens on the row numbers in
%% Qs.
safe(Q,Qs) ->
  safe(Q,1,Qs).

safe(_Q,_I,[]) ->
  true;
safe(NewQ,I,[Q|Qs]) ->
  NewQ /= Q andalso NewQ /= Q+I andalso NewQ /= Q-I andalso safe(NewQ,I+1,Qs).

benchmark(F,N) ->
  Answer = queens(N),
  Times = [begin
             {T,Ans} = timer:tc(?MODULE,F,[N]),
             if Ans==Answer ->
                 io:format("."),
```

```
                T;
               true ->
                exit(wrong_answer)
            end
          end
          || _ <- lists:seq(1,20)],
  io:format("\n"),
  lists:sum(Times)/20000.
```

`queens:queens(N)` returns a list of all such solutions, using the representation given above. For example,

```
> queens:queens(8).
[[4,2,7,3,6,8,5,1],
 [5,2,4,7,3,8,6,1],
 [...]|...]
```

The `benchmark(F,N)` tests a solving function called `F` in the current module, calling it at size $\texttt{N} \times \texttt{N}$, comparing its result to `queens(N)`, and returning the average execution time of 20 calls, in milliseconds. Solving the 12×12 problem using the `queens` function should take around 20 seconds.

```
> queens:benchmark(queens,12).
....................
845.29305
```

Copy this code into `queens.erl`, compile it, and run the tests above.

We are going to develop functions to compute the solutions in parallel, and add them to this file. *Do not change the code that is already there!* Add this alternative function to the file:

```
queens2(N) ->
  queens2([],N,N).

%% Given a partial solution Qs, containing a valid placement of N-M
%% queens on an (N-M)xN chessboard, queens2(Qs,M,N) returns all
%% solutions obtained by adding M more queens to the left, resulting
%% in a list of solutions on an NxN chessboard.
queens2(Qs,0,_) ->
  [Qs];
queens2(Qs,M,N) ->
  [Solution
   || Q <- lists:seq(1,N),
      safe(Q,Qs),
      Solution <- queens2([Q|Qs],M-1,N)].
```

Now benchmark it as follows:

```
> queens:benchmark(queens2,12).
....................
798.44085
```

You will add parallel functions to the file, and benchmark them, in the same way; then you will submit the resulting `queens.erl` file as part of your exam answer. You should also submit a file `Q3.txt` containing the rest of your answer, with each part clearly labelled.

(a) Study the code of `queens` and `queens2`. We aim to construct a parallel solution based on one of these functions, by performing recursive calls in parallel. Which function will you choose as a starting point for parallelization, `queens` or `queens2`? Why? **3 points**

(b) Write a new function `queens3` by copying either `queens` or `queens2`, and modifying the code so that the recursive calls in the auxiliary function are performed in new parallel processes. Add your code to `queens.erl`. **4 points**

(c) Test your code on the eight-queens problem, using `queens:benchmark(queens3,8)`. If you see a `wrong_answer` exception, then your code is wrong—fix it before continuing. Compare the benchmark results to the sequential function you started from. Copy the results into `Q3.txt`. Is your parallel code faster or slower than the sequential version? Why? **2 points**

(d) Test your code on the 12-queens problem using `queens:benchmark(queens3,12)`. You will probably encounter a problem—if you do not, try 13 queens, 14 queens, and so on, until you do. What goes wrong, and why does this happen? **2 points**

(e) Explain why *task granularity* is important in parallel programming. **2 points**

(f) How would you go about *increasing the task granularity* in your `queens3` function? Do not write code yet, just explain your intended approach. **1 point**

(g) Write a new parallel search function `queens4` (by copying and modifying `queens3`), which uses your idea from part 3f. Add the code to `queens.erl`. **4 points**

(h) Benchmark your new function using

```
queens:benchmark(queens4,12)
```

Try three different granularities. Copy the benchmark results into `Q3.txt`, clearly labelled.

i. Is your best solution faster or slower than the sequential function you started from? By how much? **1 point**

ii. Which granularity gave the best performance? **1 point**

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

This Haskell code implements (a sequential version of) Map-Reduce:

```
module MapReduce where

mapReduce :: Eq k' =>
  ((k, v) -> [(k', v')]) -> ((k', [v']) -> (k',v')) -> [(k, v)] -> [(k',v')]
mapReduce m r kvs =
  map r (groupByKey [(k',v') | (k,v)<-kvs, (k',v') <- m (k,v)])

groupByKey ((k,v):kvs) =
  (k,v:[v' | (k',v')<-kvs, k==k']) : groupByKey [(k',v') | (k',v')<-kvs, k/=k']
groupByKey [] = []
```

(a) Given the following test data,

```
test :: [(Int,[String])]
test = [(10,["hello","clouds"]),(45,["hello","sky"])]
```

which represents words occuring on (numbered) pages, define functions `mapper1` and `reducer1` to pass to `mapReduce`, to count the number of occurrences of each word in the input. That is, your code should behave like this:

```
*MapReduce> mapReduce mapper1 reducer1 test
[("hello",2),("clouds",1),("sky",1)]
```

(The order of elements in the resulting list is not important). Copy your code into `Q4.txt` together with your other answers; you do not need to submit an executable Haskell file for this question. **1 point**

(b) Define functions `mapper2` and `reducer2` so that `mapReduce` computes an *index*, recording for each word the page numbers on which it appears:

```
*MapReduce> mapReduce mapper2 reducer2 test
[("hello",[10,45]),("clouds",[10]),("sky",[45])]
```

1 point

(c) Suppose a distributed MapReduce computation is to be divided into ten map jobs and ten reduce jobs. What *communication pattern* should we expect between these jobs? Which of the following statements is true? **1 point**

i. Map jobs communicate with each other, and reduce jobs communicate with each other, but map jobs need not communicate with reduce jobs.

ii. Each map job communicates only with its corresponding reduce job.

iii. Every map job communicates with every reduce job.

(d) A MapReduce cluster consists of a collection of networked computers, supporting a distributed file system.

i. What is the difference between a *local* file and a *replicated* file in such a file system? **1 point**

ii. What kind of file (local or replicated) is used to hold the output of a map job? A reduce job? **1 point**

(e) If a node in the cluster crashes, some jobs run on that node need to be restarted on other nodes. Which jobs need to be restarted in this case?

i. The jobs running on that node at the time of the crash?

ii. All the jobs ever run on that node during the current MapReduce computation, including those that were completed before the crash?

iii. All the map jobs ever run on that node during the current MapReduce computation, and all the reduce jobs running on the node at the time of the crash?

Justify your answer. **1 point**