CHALMERS TEKNISKA HÖGSKOLA
Dept. of Computer Science and Engineering

08:30–12:30, Friday, June 5, 2020.
Parallel Functional Programming
DAT280, DIT261

Exam in Parallel Functional Programming

08:30–12:30, Friday, June 5, 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. If you have extended time, let the Examiner (Mary) know in advance and submit exactly as above, and within your extended time period.

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

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

(b) What is the main advantage of the *Strategies* approach to parallel programming in Haskell? **1 point**

(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 point**

(d) 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**

(e) Haskell and Erlang both use *garbage collection* to recycle memory, but they work rather differently. What aspect of garbage collection may cause a problem in real-time systems, and how does Erlang's VM design mitigate that problem? **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	`[{a,2},{b,2},{c,1}]`
3	`[{b,1},{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. **Scalability** **2 points**

The following Erlang code implements a simple server whose purpose is just to deliver a different number every time `unique(Server)` is called.

```
-module(unique).
-export([unique_server/0,unique/1).

unique_server() ->
  spawn_link(fun() ->
    unique_server(0)
  end).

unique_server(N) ->
  receive
    {get_unique,Pid,Ref} ->
      Pid ! {unique,N,Ref},
      unique_server(N+1)
    end.

unique(Server) ->
  Ref = make_ref(),
  Server ! {get_unique,self(),Ref},
  receive {unique,N,Ref} -> N end.
```

For example, a sample run in the Erlang shell might be as follows:

```
11> Server = unique:unique_server().
<0.56.0>
12> [unique:unique(Server) || _ <- lists:seq(0,10)].
[0,1,2,3,4,5,6,7,8,9,10]
```

Suppose that profiling a system whose performance is poor shows that the unique number server has very many messages in its mailbox.

(a) How would you interpret the large number of messages in this mailbox? **1 point**

(b) Suggest a way to mitigate this scalability problem. (Do not write code, just explain your idea). **1 point**

3. **Work and Depth (and control of granularity)** **16 points**

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

(a) A sequential left fold operating on a list in Haskell operates on the elements of the list one after another, from left to right. What are the work and depth (or span) of such a fold? Explain your answer. **2 points**

(b) When we have an associative operator, we can instead make use of parallelism and use a tree shaped fold (often called *parallel reduce*). Explain why the associativity of the operator enables the use of parallelism. **2 points**

(c) In Haskell, using the `Par` monad, complete the definition of the following parallel reduction function: **3 points**

```
parReduce :: NFData a => (a -> a -> a) -> [a] -> Par a
parReduce f [a] = ...
parReduce f as  =
     let  halfn = div (length as) 2 in
     let  (ys,zs) = splitAt halfn as in
     do
       i <- spawn $ ...
       ...
       return ...
```

(d) What are the work and depth of the new tree shaped reduce? **2 points**

Would you expect good performance from the above function? Explain your answer. **1 point**

Whatever you answered, modify the function in two different ways to enable control of task granularity.

i. Add a depth parameter and revert to a sequential fold when it reaches zero. **2 points**

ii. Chunk the input data and run several sequential folds in parallel, before completing the computation. (Hint: You may wish to make use of the strict sequential fold, `foldl1'` provided in `Data.List`. You may simply use `parMap` without defining it.) **2 points**

iii. Benchmark both versions and report on performance. Use an input list size that is somehow related to the last four digits of your personal number. If you don't have a personal number, make up four digits! (Hint: Be careful not to spend too much time on this if things go wrong. You will need to dream up a reasonably expensive binary operator. `(+)` is too small, for instance. Indicate whether or not your operator is actually associative.) **2 points**

4. **Scan** **12 points**

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

(a) 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});
```

(b) A naive implementation of the Blelloch scan (or similar) in Haskell would perform poorly due to lack of control of task granularity. We will here explore a chunking approach.

i. A seqential scan has an "accumulator" that makes it difficult to do simple chunking. One way forward is to first make a useful building block in the form of a sequential scan that explicitly takes an accumulator.

```
scanAcc f (m,(x:xs)) = scanl1 f ((f m x):xs)

*Main> scanl1 (+) [1..5]
[1,3,6,10,15]

*Main> scanAcc (+) (3,[1..5])
[4,6,9,13,18]
```

Now, we would like to implement a scan using `parMap` of `scanAcc`. But the question is how do we figure out what the first inputs to the calls of `scanAcc` should be, and how can we compute them as cheaply as we can, and in parallel? You are asked to solve this problem and write a chunking implementation of parallel scan by completing the following definition (or otherwise): **4 points**

```
parScanChunk :: (Num a, NFData a) => Int -> (a -> a -> a) -> [a] -> Par [a]
parScanChunk n f as = let aas = chunk n as in
                          do ms <- ... aas
                             ...  parMap (scanAcc f) ...
```

(Hint: I have added a `Num a` constraint so that the accumulator for the leftmost `scanAcc` can be `0`. Ignore this if you wish. So what should be the first inputs to all the other `scanAcc`s? Think about how the last element of the output of a scan is computed. Ignoring parallelism for a moment, the following examples may be helpful.

```
*Main> map (scanl1 (+)) . chunk 8 $ [1..100]
[[1,3,6,10,15,21,28,36]
,[9,19,30,42,55,69,84,100]
,[17,35,54,74,95,117,140,164]
,[25,51,78,106,135,165,196,228]
,[33,67,102,138,175,213,252,292]
,[41,83,126,170,215,261,308,356]
,[49,99,150,202,255,309,364,420]
,[57,115,174,234,295,357,420,484]
,[65,131,198,266,335,405,476,548]
,[73,147,222,298,375,453,532,612]
,[81,163,246,330,415,501,588,676]
,[89,179,270,362,455,549,644,740]
,[97,195,294,394]]
```

```
*Main> scanl1 (+) [1..100]
[1,3,6,10,15,21,28,36
,45,55,66,78,91,105,120,136
,153,171,190,210,231,253,276,300
,325,351,378,406,435,465,496,528
,561,595,630,666,703,741,780,820
,861,903,946,990,1035,1081,1128,1176
,1225,1275,1326,1378,1431,1485,1540,1596
,1653,1711,1770,1830,1891,1953,2016,2080
,2145,2211,2278,2346,2415,2485,2556,2628
,2701,2775,2850,2926,3003,3081,3160,3240
,3321,3403,3486,3570,3655,3741,3828,3916
,4005,4095,4186,4278,4371,4465,4560,4656
,4753,4851,4950,5050]
)
```

ii. Benchmark your function for various chunk sizes, again with an input list whose length is related to the last four digits of your personal number, and comment on the results. **2 points**

(c) Now, let us think a bit more abstractly about work and depth! Studying work (the time taken on one processor, T_1) and depth (the time taken on an infinite number of processors, T_∞) gives us insights about the time taken on n processors, T_n.

i. Explain in your own words what we know of the form $T_n \geq$??? **2 points**

ii. Explain in your own words what we know of the form $T_n \leq$??? (via Brent's theorem). **2 points**

5. **Parallel Erlang Programming** **20 points**

Copy the following code into `pmap.erl`, and compile it.

```
-module(pmap).

-compile(export_all).

map(_,[]) -> [];
map(F,[X|Xs]) -> [F(X)|map(F,Xs)].

%% Test code--do not touch

test(F) ->
  Case = fun(N) -> [rand:uniform(200) || _ <- lists:seq(1,N)] end,
  Test = fun(N) -> Xs = Case(N),
                   {T1,Ys} = timer:tc(fun()->map(fun fac/1,Xs) end),
                   {T2,Zs} = timer:tc(fun()->?MODULE:F(fun fac/1,Xs) end),
                   if Ys==Zs ->
                       [io:format("Speedup at size ~p: ~px\n",[N,T1/T2])
                        || T2/=0];
                      true ->
                       io:format("Buggy!\n  ~p\n/=\n  ~p\n",[Zs,Ys]),
                       exit(buggy)
                   end,
                   T1+T2<5000000
         end,
  Test(10)
    andalso Test(100)
    andalso Test(1000)
    andalso Test(3000)
    andalso Test(10000)
    andalso Test(30000)
    andalso Test(100000)
    andalso Test(300000)
    andalso Test(1000000)
    andalso Test(3000000)
    andalso Test(10000000).

fac(0) -> 1;
fac(N) -> N*fac(N-1).
```

This code defines a sequential version of the `map` function, along with some test and benchmarking code. Run the tests as follows:

```
147> pmap:test(map).
Speedup at size 100: 0.0x
Speedup at size 3000: 1.9375x
Speedup at size 10000: 0.9841269841269841x
Speedup at size 30000: 0.9999950738916256x
Speedup at size 100000: 0.9781644514766397x
Speedup at size 300000: 1.0073278110948922x
Speedup at size 1000000: 0.9788467640395393x
false
```

Calling `pmap:test(Name)` tests a function called `Name` in the `pmap` module, checking that its results are the same as `map` returns, and displaying a table of speedups for inputs of varying sizes. Here you see example output when the function we test is `map` itself: of course its results are correct, and the

measured "speedup" is close to 1. The benchmarks may take up to about 20 seconds to run, and the measured speedups are not very accurate—don't worry about this, we just don't have time in an exam to run enough benchmarks to make very accurate measurements.

In this question, you will extend this file with several parallel implementations of `map`. You should submit your final version of the file as part of your answer. Each new function you write can be based on the previous one; copy-and-paste the previous function, rename it, and work from there. Make sure that you *keep* every function in your file, and *separate* the parts of your answer clearly in the file—for example, as follows:

```
%% ======== part (b) ========================================
```

Make sure the file you submit is compileable, and that we can run these tests for the functions you write. Place the other parts of your answer in a file called `Q5-answer.txt`.

Hint: You may find the functions `seq`, `sort`, `split` and `zip` from the `lists` module useful in answering this question. Documentation of these functions can be found here: `https://erlang.org/doc/man/lists.html`.

(a) The code you have been given for `map` is entirely sequential. Use Erlang's parallel programming features to define a function `pmap(F,Xs)` which computes the same result as `map(F,Xs)`, but performs the calls to `F` in parallel (running each call to `F` in a different process). Make sure your `pmap` returns the elements of the result in the same order as `map`, by using selective receive to receive the elements in the correct order. Add your definition of `pmap` to `pmap.erl`, and test it using `pmap:test(pmap)`. (Do not change the test code: this will compare your new implementation to the existing `map` implementation). Copy-and-paste the output of your test run into `Q5-answer.txt` (clearly labelled).

Does your parallel code run faster or slower than the sequential `map`? If it runs slower, how would you explain this? **4 points**

(b) Another way to ensure the elements of your result are in the correct order is to pair each one with its index in the list, receive them in any order, and sort them after receipt, using `lists:sort` (which orders pairs lexicographically, so that `{I,X} < {J,Y}` if `I < J`, or `I==J` and `X < Y`). Add a new parallel map function to your file, `pmap2`, which works in this way. Test your function using `pmap:test(pmap2)`, and copy-and-paste the output into `Q5-answer.txt`.

Is there a significant performance difference between `pmap` and `pmap2`? If so, how would you explain it? **4 points**

(c) Why is *task granularity* important in parallel programming? Explain the problems that can arise both when task granularity is too large, and when it is too small. **2 points**

(d) Is the task granularity in your implementation of `pmap2` likely to be too large, just right, or too small? Why? **1 point**

(e) Add another parallel version of `map` to `pmap.erl`, called `pmap3`, which uses a larger task granularity, performing ten calls to `F` in each process. Run the tests, and copy-and-paste the output into `Q5-answer.txt`. How does performance change? **4 points**

(f) What advantages might there be in *limiting* the number of parallel processes working on a task at the same time? **1 point**

(g) Add a fourth parallel version of `map` to your file, called `pmap4`, which uses a larger task granularity just like `pmap3`, but also limits the number of simultaneously running worker processes to the number of hardware threads available (which you can find using `erlang:system_info(schedulers)`). Test your function using `pmap:test(pmap4)`, and copy-and-paste the results into `Q5-answer.txt`. How does performance compare to `pmap3`? **4 points**