

Deterministic Parallel Programming in Haskell

Feel free to do the exercises in any order, or to adapt them to your liking.

Exercise 1. Try to distinguish the partially defined Boolean lists

```
undefined          :: [Bool]
undefined : undefined :: [Bool]
undefined : []      :: [Bool]
True             : undefined :: [Bool]
```

using suitable strategies. To get the default strategies, you have to import the module `Control.Parallel.Strategies`.

Exercise 2. Define your own `NFData` instances for lists, pairs, and `Tree`:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
deriving (Eq, Ord, Show)
```

Note that once you import `Control.DeepSeq`, you automatically have the standard instances. So you might want to define your own datatypes isomorphic to lists and pairs, and instantiate these.

Exercise 3. Reproduce the results from the slides using the `ParTest` example programs. Try to use `ThreadScope` to visualize the event logs. Remember to pass the `-1s` RTS option to the actual program to generate an eventlog, and a suitable `-N` option to get parallel behaviour.

Exercise 4. The file `ParTest3A` contains a variant of `ParTest3` that looks simpler, but behaves much worse. Can you explain why?

Exercise 5. The file `ParTest6A` contains a variant of `ParTest6` that might also seem reasonable, but behaves much worse (for two cores, it may look ok, but for more, it doesn't). Can you explain why?

Exercise 6. Try to replace the `collatz` function with another function. What happens if you choose a trivial (cheap) function? What happens if you choose a very memory-intensive function, such as the naive definition of the Fibonacci numbers?

Exercise 7. Generalize the `mapReduce` skeleton slightly, so that it isn't tied anymore to problems that have integer bounds. Remember, however, that lists are expensive to split! So find a generalization that at least not forces you to go via lists. Verify that the `collatz` example still works.

Exercise 8. Try to parallelize the n-queens problem (described on slides, contained in `Queens.hs`). The key to parallelization is that the given algorithm is essentially a brute-force search. In each step, we try to place a new queen in all safe positions. Instead of exploring the search tree linearly and depth-first, we can as well try to explore it in parallel up to a certain threshold.

Exercise 9. Can you extract a useful skeleton from your parallelization of n-queens?

Exercise 10. Pick your own real-world problem and try to parallelize it in Haskell.