Outline of Lecture 15

- Parallel programming in Haskell
- Lazy evaluation and parallel evaluation (the `Eval` monad)
- Dataflow programming (the `Par` monad)
- Functional programming and its applications outside Haskell
- Exam information. The topics for exam exercises

Parallel Haskell

- Parallel Haskell is aimed at providing access to multiple processors in a natural and robust way
- **Advantage**: parallel programming in Haskell is deterministic (thus, adding parallelism leads to no race conditions or deadlocks)
- **Advantage**: Haskell program are high level and declarative
  - more likely to work on wide range of parallel hardware
  - by default, take advantages of the current highly tuned technologies (like parallel garbage collection) and their future advancements
- **Disadvantage**: A lot of execution details are hidden $\Rightarrow$ performance problems can be sometimes hard to understand

Parallel Haskell (cont.)

- The main thing that the parallel Haskell programmer has to think about is **partitioning** the problem into parallel computations
- Two sub-issues
  - **Granularity:** not too high, not too low to keep the processors busy. Depend on static or dynamic partitioning
  - **Data dependencies:** when one task depends on another, they must be performed sequentially. Two approaches in Haskell: entirely implicit (using the `Eval` monad) or explicit (using the `Par` monad).

    The latter (explicit) one is less concise but it can be easier to understand and fix problems

Lazy evaluation and basic paralellism

- Haskell is lazy $\Rightarrow$ expressions are not evaluated until they are required
- Example:

```
Prelude> x = 1 + 2 ::  Int
```

- Semantically, it looks equivalent to `x = 3`
- Calculationally, the computation of `1 + 2` has not happened yet, so we might compute it in parallel with something else
- We say that at this point `x` is *unevaluated*
- Printing the actual value of `x` without causing it to be evaluated:

```
Prelude> :sprint x
x = _
```

Lazy evaluation and basic parallelism (cont.)

- Unevaluated, the expression `1+2` is just an object in memory (called *thunk*) and `x` is a pointer to it
- Forcing evaluation:

```
Prelude> x
3
Prelude> :sprint x
x = 3
```

Lazy evaluation and basic parallelism (cont.)

- Dependent definitions:

```
Prelude> x = 1 + 2 ::  Int
Prelude> y = x + 1
Prelude> :sprint x
x = _
Prelude> :sprint y
y = _
```

- One step of evaluation with the built-in Haskell function `seq` (evaluates the first argument and returns the second one)

```
Prelude> seq y ()
()
Prelude> :sprint x
x = 3
Prelude> :sprint y
y = 4
```

Lazy evaluation and basic parallelism (cont.)

- `seq` forces one-step evaluation, e.g., for the first constructor, and stops there:

```
Prelude> xs = map (+1) [1..10] ::  [Int]
Prelude> :sprint xs
xs = _
Prelude> seq xs ()
()
Prelude> :sprint xs
_ : _
```

- Applying `length` to `xs` (no need to evaluate the actual elements):

```
Prelude> length xs
10
Prelude> :sprint xs
xs = [_,_,_,_,_,_,_,_,_,_]
```

Lazy evaluation and basic parallelism (cont.)

- Forcing full evaluation of `xs`:

```
Prelude> sum xs
65
Prelude> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

- By using Haskell pre-defined features from the `Control` module, we can control the evaluation process: which evaluations to do in parallel or in sequence

The `Eval` monad

- The basic functionality for creating parallelism is provided by the `Eval` monad (the module `Control.Parallel.Strategies`):

```
data Eval a
instance Monad Eval

runEval ::  Eval a -> a

rpar ::  a -> Eval a
rseq ::  a -> Eval a
```

- The `rpar` combinator creates parallelism, i.e., instructs the unevaluated argument to be evaluated in parallel. Does not wait for the result before passing the control
- The `rseq` combinator forces sequential evaluation: evaluates the arguments and waits for the result
- `runEval` performs the `Eval` computation and returns its result

The `Eval` monad (cont.)

- Example:

```
runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  return (a,b)
```

- `f x` and `f y` begin to evaluate in parallel, while return happens immediately
- The rest of the program will continue to execute while `f x` and `f y` are being evaluated in parallel
- By default, only a single evaluation step (for the first encountered constructor) is done

The `Eval` monad (cont.)

- Example:

```
runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  rseq a
  rseq b
  return (a,b)
```

- `f x` and `f y` begin to evaluate in parallel
- `rseq a` and `rseq b` force to wait until the evaluation is over before returning
- Many other evaluation strategies are possible

Example: Parallelising a Sudoku Solver

- The given `Sudoku` module contains a function

  ```
  solve :: String -> Maybe Grid
  ```

  that solves a single Sudoku problem (represented as `String`). The result is `Nothing` if no solution exists

- Sequential code (sudoku1.hs) to solve a set of Sudoku problems from a file:

```
import Sudoku
...
main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f
  let puzzles = lines file
      solutions = map solve puzzles
  print (length (filter isJust solutions))
```

Example: Parallelising a Sudoku Solver (cont.)

- Compiling sudoku1.hs with full optimisation:

```
$ ghc -O2 sudoku1.hs -rtsopts
[1 of 2] Compiling Sudoku ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main ( sudoku1.hs, sudoku1.o )
Linking sudoku1 ...
```

- Running sudoku1.hs on 1000 sample problems (with extra options to get statistics):

```
$ ./sudoku1 sudoku17.1000.txt +RTS -s
1000
...
Total time 0.890s ( 0.899s elapsed)
```

Example: Parallelising a Sudoku Solver (cont.)

- Let us split the list of 1000 problems in two and run the solver in parallel on two cores
- Parallel code (sudoku2.hs):

```
main ::  IO ()
main = do
  [f] <- getArgs
  file <- readFile f
  let puzzles = lines file
      (as,bs) = splitAt (length puzzles `div` 2) puzzles
      solutions = runEval $ do
         as' <- rpar (force (map solve as))
         bs' <- rpar (force (map solve bs))
         rseq as'
         rseq bs'
         return (as' ++ bs')
  print (length (filter isJust solutions))
```

Example: Parallelising a Sudoku Solver (cont.)

- `force` – full evaluation instead of one evaluation step
- Compiling sudoku2.hs with full optimisation (threaded version):

```
$ ghc -O2 sudoku2.hs -rtsopts
[2 of 2] Compiling Main ( sudoku2.hs, sudoku2.o )
Linking sudoku2 ...
```

- Running sudoku2.hs on 1000 sample problems on two cores (option `-N2`):

```
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -s
1000
...
Total time 0.975s (0.608s elapsed)
```

- Speedup on the elapsed time (comparing to the sequential version): $0.899 / 0.608 = 1.48$

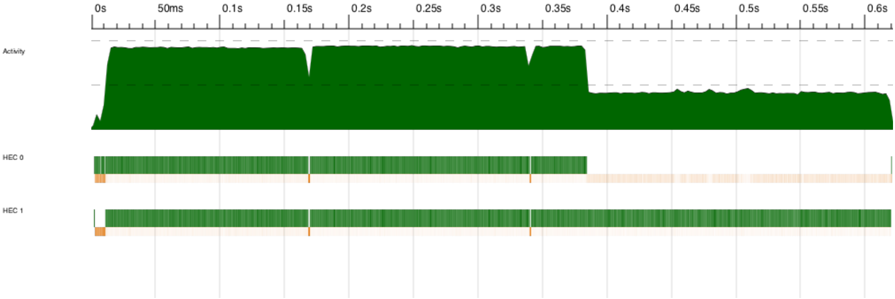
Example: Parallelising a Sudoku Solver (cont.)

- The speedup 1.48 is not bad, but could be better. Where are the losses?
- Creating an event log for analysis:

```
$ rm sudoku2; ghc -O2 sudoku2.hs -rtsopts
-threaded - eventlog
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -l
```

- Using the ThreadScope tool to analyse the log

Example: Parallelising a Sudoku Solver (cont.)



- It looks like the work load between two lists was unevenly distributed and one processor finished executing earlier and stayed idle for quite some time
- How to improve on that problem? We have to make work "chunks" (the tasks to be handled in parallel) smaller

Example: Parallelising a Sudoku Solver (cont.)

- Haskell supports *dynamic partitioning* of such work tasks given to `rpar` to the available idle processors
- We just have to create a pool with enough tasks by calling `rpar` often enough so it can do its job and balance the work evenly
- First, let's define abstraction that will let us apply a function to a list in parallel:

```
parMap ::  (a -> b) -> [a] -> Eval [b]
parMap f (a:as) = do
  b <- rpar (f a)
  bs <- parMap f as
  return (b:bs)
```

Example: Parallelising a Sudoku Solver (cont.)

- Let us apply dynamic partitioning and run the solver in parallel on two cores:
- Parallel code (sudoku3.hs):

```
main ::  IO ()
main = do
  [f] <- getArgs
  file <- readFile f
  let puzzles = lines file
      solutions = runEval (parMap solve puzzles)
  print (length (filter isJust solutions))
```

Example: Parallelising a Sudoku Solver (cont.)

- Compiling sudoku3.hs with full optimisation (threaded version):

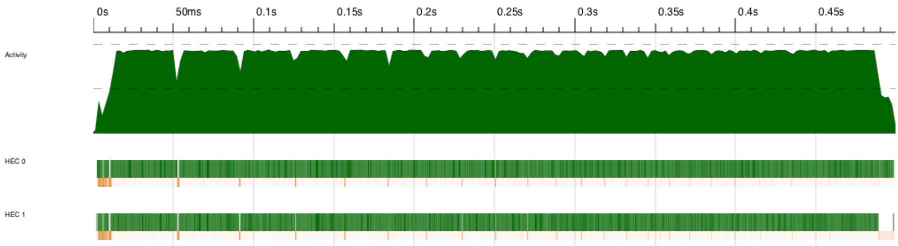
```
$ ghc -O2 sudoku3.hs -rtsopts
[1 of 2] Compiling Sudoku ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main ( sudoku2.hs, sudoku2.o )
Linking sudoku3 ...
```

- Running sudoku3.hs on 1000 sample problems on two cores:

```
$ ./sudoku3 sudoku17.1000.txt +RTS -N2 -s
1000
...
Total time 0.949s (0.490s elapsed)
```

- Speedup on the elapsed time (comparing to the sequential version): $0.899 \;/\; 0.490 = 1.83$

Example: Parallelising a Sudoku Solver (cont.)



- Much better work load distribution
- The last run: the same sudoku3.hs, but on 4 cores

```
$ rm sudoku3; ghc -O2 sudoku3.hs -rtsopts
$ ./sudoku3 sudoku17.1000.txt +RTS -N4 -s
...
Total time 0.996s (0.265s elapsed)
```

- Speedup on the elapsed time (comparing to the sequential version): $0.899 / 0.265 = 3.39$

Dataflow parallelism: the `Par` monad

- The `Eval` monad heavily relies on lazy evaluation to express parallelism, allows decoupling of the algorithm from parallelism as well as building different parallel strategies compositionally
- Another parallel programming model, the `Par` monad, is more explicit about granularity and data dependencies. We have to give more details, but in return gain more control
- Implements dataflow parallelism (dataflow networks)

Dataflow parallelism: the `Par` monad (cont.)

- The interface:

```
newtype Par a
instance Monad Par

runPar ::  Par a -> a
fork ::  Par a -> Par a
```

- A computation in the `Par` monad can be run using `runPar` to return a pure result
- Parallelism is introduced by `fork`, the purpose of which is to create parallel tasks
- The `Par` computation passed as as the `fork` argument ("the child") is executed in parallel with `fork` caller

Dataflow parallelism: the `Par` monad

- `fork` does not return anything to the caller. How can we get a result back from a created parallel computation?
- Data can be passed between `Par` computations using the `IVar` type and its operations:

```
data IVar a

new ::  Par (IVar a)
put ::  NFData a => IVar a -> a -> Par a
get ::  IVar a -> Par a
```

- Here, `NFData a` (*normal-form data*) – type class of fully evaluated data, i.e, values with no unevaluated expressions

Dataflow parallelism: the Par monad

- We can think of IVar value as a box that starts empty (after the operation new)
- The put operation stores a value in the box, while the get operation reads the value
- If get finds the box empty, it waits until its is filled by put. The box can be written only once
- So IVar lets us communicate values between parallel Par computations, because we can put a value in the box in one place and get in another

Dataflow parallelism: simple example

- Two independent computations in parallel (calculating Fibonacci values for the distinct arguments m and n)

```
runPar $ do
  i <- new
  j <- new
  fork (put i (fib n))
  fork (put j (fib m))
  a <- get i
  b <- get j
  return (a+b)
```

Functional programming in other languages

- Increasing number of functional programming features in the current imperative languages (Python, Java, JavaScript, C#): making functions into "first class citizens", constructing functions on-the-fly with lambda notation, functional composition, immutability
- Synergy of FP and OOP in Scala
- Java: a lot of new functional features in Java 8
- Extra material: FP in Java 8, FP and Scala

Applications of functional programming in real world

- Frameworks for Big Data and data analytics (Hadoop Map Reduce, Cascalog, Apache Spark, R Studio, etc.): chaining functional computations, combining and transforming data sets
- Building interactive fault-tolerant concurrent systems and NoSql database systems with Erlang: using immutable data, pattern matching, functional programming for interacting processes exchanging data messages
- Extra material: FP in R, FP and Erlang

Exam information

- Exam date: January 12th, 12.00 - 14.00, Didlaukio 102 auditorium
- 4-6 small Haskell tasks (slightly easier than homework exercises)
- Must-know exam task topics:
  - Pattern matching; Guards; Local definitions
  - Primitive recursion;
  - Working with lists, including list comprehensions;
  - Higher order functions (such as `map`, `fold`, `filter`), function composition and partial function application;
  - User-defined datatypes;
  - Haskell type classes (no functors or monads!).