WS 2024/2025

Advanced Functional Programming

Week 10 – Introduction to Parallelism and Concurrency

René Thiemann

Department of Computer Science

Last Week

- lazy I/O, file access via handles
- spawning external processes
- communication via (temporary) files
- communication via pipes with interactive processes
- exception handling
  - throw everywhere, catch in `IO`-monad
  - force evaluation, so that `try` and `catch` have an effect

Parallelism and Concurrency

- parallelism
  - aim: speed up some computation by using multiplicity of computational hardware (multicore CPU, GPU, multiprocessor machine, ...)
  - effect of using multiple cores is visible in execution time, but not on result
  - example: parallel sorting algorithm, parallel matrix-multiplication algorithm, ...
- concurrency
  - program structuring technique with multiple threads of control
  - threads are executed at the same time (interleaved or on multicore systems)
  - effects of interleaving are visible
  - example: webserver has separate thread for user interface, and spawns separate threads for each download
  - example: termination prover for TRSs tries several termination techniques in parallel threads and takes result of first successful technique
- Haskell offers support for both parallelism and threads

Introduction to Parallelism

Parallelism in Haskell

- for understanding parallelism in Haskell it is crucial to understand Haskell's lazy evaluation strategy
- situation is very similar to exception handling
- both `try` and parallel evaluation should somehow enforce evaluation within the try-block, or within the parallel execution block
- bad example with try:
  ```
  let p = try (return (f x, f y)) in p
  ```
  this code will not evaluate `f x` and `f y` within the `try`-block due to lazy evaluation
- bad example with parallelism:
  ```
  let p = runEval (rpar (f x, f y)) in p
  ```
  this code will not evaluate `f x` and `f y` in parallel due to lazy evaluation
- last week: use DeepSeq to enforce full evaluation to normal form
- upcoming: more fine-grained control how to evaluate expressions

Inspecting Evaluation with :sprint

- recall
  - by default, evaluation of expressions is only trigged on demand
  - using `seq`, one can enforce evaluation to WHNF (outermost constructor)
  - using `force` of DeepSeq, one can enforce evaluation to full normal form
- with `ghci` command `:sprint expr` one can observe current evaluation status
- example

```
ghci> let xs = map (+1) [1 .. 10 :: Int]
ghci> :sprint xs
xs = _                          -- _ represents a thunk: not yet evaluated
ghci> seq xs () -- or: null xs
()              -- or: False
ghci> :sprint xs
_ : _
```

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

```
ghci> seq (force xs) ()    -- or: sum xs
()                         -- or: 65
ghci> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

Parallelism via Control.Parallel.Strategies

- this module lets user design a strategy how to evaluate expressions

  ```
  data Eval a  -- not revealed
  instance Monad Eval

  runEval :: Eval a -> a

  rpar :: a -> Eval a
  rseq :: a -> Eval a
  ```
- parallelism is expressed via `Eval` monad
- `rpar` creates parallelism
  - `rpar expr` says that `expr` should be evaluated, perhaps in parallel
  - argument to `rpar` should be a thunk (otherwise, no work needs to be done)
- `rseq` enforces sequential evaluation: wait until argument is evaluated
- both `rpar` and `rseq` refer to WHNF in evaluation
- the `r` in `rpar` and `rseq` refers to `r`ewrite to WNHF (in parallel or sequential)

Examples

- we assume that `f` is some costly operation

```
runEval $ do { a <- rpar (f x); b <- rpar (f y); return (a, b) }   (1)
runEval $ do { a <- rpar (f x); b <- rseq (f y); return (a, b) }   (2)
runEval $ do { a <- rpar (f x); b <- rpar (f y);                   (3)
               rseq a; rseq b; return (a, b) }
```

- in (1), the return happens immediately; remaining program continues evaluation while `f x` and `f y` are evaluated in parallel
- in (2), the return happens after `f y` has been evaluated to WHNF; evaluation of `f x` and `f y` happen in parallel, and evaluation of `f x` continues in parallel after return
- in (3), the evaluation of `f x` and `f y` are in parallel; however, the return is only executed after both `f x` and `f y` are evaluated to WHNF

Running the Examples

- we test the previous example with f = fib, x = 37, y = 35

```
mainFib n = do
  let test = [test1, test2, test3] !! (read n - 1)
  t0 <- getCurrentTime
  r <- evaluate (runEval test)
  printTimeSince t0                -- return time
  print r
  printTimeSince t0                -- full evaluation time
```

- running parallel programs requires
  - compilation with `-threaded` flag
  - execution with `+RTS -Nn -RTS` where `n` is maximal number of cores
  - example: run test 1 with at most 2 cores via cabal:
    `cabal run Demo10 -- fib 1 +RTS -N2 -RTS`
- execution times

```
n = 1: 0.0s, 0.47s  (1)    0.19s, 0.47s  (2)     0.47s, 0.47s  (3)
n = 2: 0.0s, 0.30s  (1)    0.19s, 0.30s  (2)     0.30s, 0.30s  (3)
```

Parallelization of Quicksort

- consider sequential quicksort (without randomization)

```
qsortSeq (x : xs) = let
  (low, high) = partition (< x) xs
  sLow = qsortSeq low
  sHigh = qsortSeq high
 in sLow ++ [x] ++ sHigh
qsortSeq [] = []
```

- integrate parallelization: evaluate both recursive invocations in parallel
- setup for evaluating effect of parallelization
  - read list of 5 million random numbers from file (generated by `Demo10 numbers 5000000`)
  - force that reading is fully completed by using `force` from DeepSeq
    (so reading from file and parsing is done purely sequentially)
  - start timing
  - run sorting algorithm and print length of sorted list
  - stop timinig

Setup in Haskell

```haskell
sortAlgs :: [(String, [Int] -> [Int])]
sortFile :: FilePath

mainSort :: String -> IO ()
mainSort algName = do
  case lookup algName sortAlgs of
    Nothing -> error $ "unknown sorting algorithm"
    Just sortAlg -> do
      input <- lines <$> readFile sortFile
      let numbers = force $ map read input
      putStrLn $ "We have " ++ show (length numbers) ++ " elements to sort."
      start <- getCurrentTime
      let sorted = sortAlg numbers
      putStrLn $ "Sorted all " ++ show (length sorted) ++ " elements."
      end <- getCurrentTime
      putStrLn $ show (end `diffUTCTime` start) ++ " elapsed."
```

Parallelized Version of Quicksort – Try 1

- code of parallel quicksort, version 1

```
qsortPar1 (x : xs) = let
  (low, high) = partition (< x) xs
 in runEval $ do
      sLow <- rpar $ qsortPar1 low
      sHigh <- rpar $ qsortPar1 high
      rseq $ sLow
      rseq $ sHigh
      return $ sLow ++ [x] ++ sHigh
qsortPar1 [] = []
```

- time sequential: 8.39 seconds
- time parallel (-N1): 8.77 seconds
- time parallel (-N2): 5.89 seconds
- time parallel (-N4): 5.20 seconds

Observations

- minimal overhead in making algorithm parallel
  - no I/O required
  - no explicit creation of threads, etc.
  - no explicit synchronization, communication, etc.
  - no detection of finalized computations
- debugging of parallel code can done by running it sequentially (not: runtime analysis)
- remark: Haskell gives no guarantee on how parallelization is executed
  - quicksort on test input invokes `rpar` a million times
  - spawning a thread for each of this invocations would be far too expensive (overhead of thread creation)
  - instead the argument to `rpar` is called a spark
  - sparks are cheap to create and are stored in a pool
  - whenever there is a spare core available, it starts to evaluate some sparks
  - overhead of spark handling is small:
    8.39 seconds (sequential algorithm) vs. 8.77 seconds (parallel algorithm with 1 core)
- algorithm is not optimal, since parallelization stops after evaluation to WHNF, i.e., after first element of recursive calls has been determined

Parallelized Version of Quicksort – Try 2

- code of parallel quicksort, version 2

```
spine (_ : xs) = spine xs
spine [] = ()
... runEval $ do
          sLow <- rpar $ qsortPar2 low
          sHigh <- rpar $ qsortPar2 high
          rseq $ spine sLow
          rseq $ spine sHigh
          return $ sLow ++ [x] ++ sHigh
```

- only difference, use `spine` to force evaluation of list structure
- effect: both recursive calls are fully evaluated in parallel
- time parallel (-N1) shows overhead of `spine`: 9.45 seconds
- time parallel (-N4) shows improved parallelization: 4.88 seconds
- note: using `force` instead of `spine` would slow down the computation, since `force` also ensures that all list arguments are fully evaluated

Parallelized Version of Quicksort – Version 3

- although overhead of sparks is small, there is some overhead
- in particular it does not pay off to run quicksort in parallel when recursion reaches small lists
- problem of granularity: divide work into reasonable chunks that are solved in parallel
  - too large chunks: several cores might become idle
  - too small chunks: overhead for each spark becomes more significant
- parallel quicksort version 3 uses simple depth limit to switch to sequential version

```
qsortPar3 = qsortPar3Main 10
qsortPar3Main d xs
  | d == 0 = qsortSeq xs
qsortPar3Main d (x : xs) = let
  (low, high) = partition (< x) xs
 in runEval $ do
      sLow <- rpar $ qsortPar3Main (d-1) low
      sHigh <- rpar $ qsortPar3Main (d-1) high
      rseq $ spine sLow                       ... 4.50 seconds
```

Final Remarks on Parallelization

- there is a lot more to explore, e.g., to have more control over parallelization via strategies or via explicit forks of sparks and dataflow parallelism
- strategies in brief
  - separate what is computed to how it is evaluated
  - examples: in the timing code, replace line

    ```
    let numbers = force $ map read input
    ```

    by the following one to get a parallel map

    ```
    let numbers = force $ (map read input `using` parList rseq)
    ```
- note that while sparks are cheap to create, beware on how data is distributed
  - without the `force` in the definition of `numbers`, there might be dependent thunks in the input list which are distributed over several cores and trigger a ping-pong effect: evaluating parts of the input on one core has to ask an evaluation of another core, etc.
  - result without `force`: sorting takes 19.41 seconds with 4 cores

Introduction to Concurrency in Haskell

Concurrency

- concurrent Haskell: facilities of Haskell for programming with multiple threads of control
- threads run independently concurrently
  - execution in parallel on multiple cores,
  - execution using time-slicing via some scheduling algorithm, or
  - combined algorithm
- threads may be put to sleep and waked up at any time
  - by scheduling algorithm (Haskell runtime or OS)
  - if some shared resource is occupied or is getting available
- overhead of thread-creation, scheduling, etc. is small (lightweight threads), but not as small as creating sparks in previous section
- viewpoint of concurrency in Haskell
  - concurrency permits us to increase modularity, e.g. separate threads for different tasks
  - Haskell provides simple, but versatile features for concurrency
    - user can stay at low-level interface to tune performance
    - user can program more high-level abstractions
- here: start with low-level interface, show how to advance to higher-level interfaces

A First Concurrent Program

- start with: `cabal run Demo10 -- td1 +RTS -N2 -RTS`

  ```
  mainThreadDemo1 = do
    hSetBuffering stdout NoBuffering
    forkIO (replicateM_ 100000 (putChar 'a'))    -- ThreadId is ignored
    replicateM_ 100000 (putChar 'b')
  ```
- buffering is turned off so that printing is immediate
- `forkIO :: IO () -> IO ThreadId`

  `forkIO a` spawns a new thread that executes action `a`,

  the current thread gets an identifier to the thread (similar to process handle)
- output is similar to `bbbbbababababababababababababababababaababababab...`
  - most of the time strict alternation of `a` and `b`
  - reason: fairness when trying to access shared resource `stdout`

A Second Example: Reminders

- start with: `cabal run Demo10 td2`

```
mainThreadDemo2 = do
  s <- getLine
  if s == "exit"
    then return ()
    else do
      forkIO $ setReminder s
      mainThreadDemo2

setReminder s = do
  let t = read s :: Int
  putStrLn $ "Reminder in " ++ show t ++ " seconds"
  threadDelay $ 10^(6 :: Int) * t
  putStrLn $ "Reminder of " ++ show t ++ " seconds is over! \BEL"
```

- `threadDelay :: Int -> IO ()` puts current thread to sleep (number of microseconds)

Observations

- when typing `"exit"`, the initial thread is done
- if this happens, the runtime system stops the complete program, i.e., also all running reminder-threads are terminated
- hence, the starting thread has a special role
  - termination of a spawned thread (any of the reminder-threads) does not lead to termination of the complete program
- note: this effect does not show up when running `mainThreadDemo2` within `ghci`

Communication: MVars

- most basic primitive to communicate via threads is via some `MVar`

```
data MVar a  -- not revealed
newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

- an `MVar a` is similar to `Maybe a`:
  it is a box that can store one value of type `a` or nothing
- the `newXXX` operations create an empty or full `MVar`
- the thread first waits (blocks) until there is a value in the `MVar`,
  and then removes the value from the `MVar` and returns it
- similarly, `putMVar` waits until the `MVar` is empty and then stores a value in it

Simple Communication Between Threads

- pass one value between two threads

```
comm1 = do
  m <- newEmptyMVar
  forkIO $ putMVar m 'x'
  r <- takeMVar m
  print r
```

  scheduling does not matter: main thread waits until forked thread has filled `m`

- pass two values between two threads

```
comm2 = do
  m <- newEmptyMVar
  forkIO $ do { putMVar m 'x'; putMVar m 'y' }
  r <- takeMVar m
  print r
  r <- takeMVar m
  print r                  -- result: print 'x' and then 'y'
```

  single `MVar` `m` is used as a channel: multiple writer, single reader

Simple Communication Between Threads: Deadlocks

- consider situation where all threads wait on change of some `MVar`

```
comm3 = do
  m <- newEmptyMVar
  n <- newEmptyMVar
  forkIO $ do { s <- takeMVar m; putMVar n (s + 1) }
  r <- takeMVar n
  putMVar m (42 :: Int)
  print r
```

- such a situation is called a deadlock and should be avoided
- invoking `comm3` in `ghci`
  - deadlock looks like a non-terminating computation
  - abort with CTRL-C
- standalone-program (`cabal run Demo10 comm3`)
  - described deadlock w.r.t. `MVar`s results in runtime exception
  - can be used for debugging

Usages of MVar

- `MVar`s are quite basic, but also versatile
- use case 1: one-place channel
  - pass messages around threads
  - limitation: one message at a time
- use case 2: container for shared mutable state (exercise, task 3)
  - choose `a` in `MVar a` as some normal immutable data
  - thread can take `a` (and acquire a lock), and then write back the modified `a`
  - if `a = ()`, then `MVar` is just used as lock
- use case 3: building block for larger concurrent data structures (next lecture)

Use Case 1: Example Application of a Logger

- develop concurrent logging service
- for simplicity, we log to stdout, but it could be a file, a database, etc.
- logging is service in a larger application, which can be programmed independently
- closely related applications: fire-and-forget writing services to a shared resource, e.g., printer spooler
- we implement logger with the following capabilities

```
initLogger :: IO Logger
logMessage :: Logger -> String -> IO ()
logStop    :: Logger -> IO ()
```

- `logStop` is required so that logger can log all pending log-messages before stopping
  - ending main thread without invoking `logStop` would result in killing the logger

The Logger

```haskell
initLogger = do
  m <- newEmptyMVar
  let l = Logger m
  forkIO (logger l)
  return l

logger :: Logger -> IO ()
logger (Logger m) = loop where
  loop = do
    cmd <- takeMVar m
    case cmd of
      Message msg -> do
        putStrLn msg
        loop
      Stop s -> do
        putStrLn "logger: stop"
        putMVar s ()
```

```haskell
newtype Logger = Logger (MVar LogCommand)

data LogCommand =
  Message String | Stop (MVar ())

logMessage (Logger m) s =
  putMVar m (Message s)

logStop (Logger m) = do
  s <- newEmptyMVar
  putMVar m (Stop s)
  takeMVar s
```

Remarks on Logger

- datatypes reveal that logger is basically a single `MVar` that stores log commands
- application for logger can result in arbitrary sequence of log messages

```
message s i = "message " ++ show i ++ " of " ++ s

mainLogger = do
  l <- initLogger
  forkIO $ mapM_ (logMessage l . message "fork 1") [1..100]
  forkIO $ mapM_ (logMessage l . message "fork 2") [1..100]
  mapM_ (logMessage l . message "main thread") [1..100]
  logStop l
```

- depending on scheduler, not all 100 log-messages of the forked messages will materialize
- because logger can store only single message at a time, logger might become bottleneck
- fairness of `MVar` and other blocking operations:
  if some thread requests a resource and this resource is getting available infinitely often, then the thread will eventually get access to that resource

Exercises

- Task 1
  - the implementation of the quicksort wrapper currently has a significant sequential phase, namely: `input <- lines <$> readFile sortFile`
  - figure out whether this part can be made more efficient by using parallelism; to this end, implement and evaluate some of the following ideas
    - reading the file is still done sequentially, but `lines` is re-implemented in a parallel way
    - both reading and splitting the input into lines is done in parallel
- Task 2
  - instead of performing parallelization with quicksort, an alternative is to split the list into n sublists (where n is the number of cores), each sublist is sorted in parallel using sequential quicksort, and then the merge-operation of mergesort is applied
  - implement and evaluate this idea
- Task 3
  - we consider the task to create a concurrent dictionary, based on a standard immutable dictionary implementation
  - the aim is to gain efficiency by releasing `MVar`-locks early on
  - the exercise will illustrate the effect of lazy evaluation in concurrency
  - further details: see Haskell source

Literature

- Simon Marlow, Parallel and Concurrent Programming in Haskell, 2013, O'Reilly, Chapters 2 and 7
- Real World Haskell, Chapter 24
- https://hackage.haskell.org/package/parallel/docs/Control-Parallel-Strategies.html
- https://hackage.haskell.org/package/base/docs/Control-Concurrent.html