Lecture 27: Parallelism & Concurrency in Functional Languages

CSC 131
Kim Bruce

Final Exam

- Comprehensive, but emphasis on last half.
  - 24 hour take-home.
  - Pick up from CS office, 2nd floor Edmunds between 9:00 a.m. - noon and 1 p.m. - 4:30 p.m.
  - Available by Monday of exam week at 9 a.m.
  - Due 24 hours after pickup, but Friday at 4 p.m. at latest.

Parallel Haskell Constructs

- Laziness in opposition to parallel
- Any communication is side-effect
  - Must use monads
- Designed for shared-memory architectures

Light-Weight Threads

- Use forkIO :: IO( ) -> IO ThreadId
- Notice it lives in IO monad:

```
main = do
   [fileA,fileB] <- getArgs
   forkIO $ hashAndPrint fileA
   hashAndPrint fileB
```

- *where $ is low precedence function application*

- Forks off process to handle fileA, while main handles fileB.

Synchronizing?

- MVar represents locking mutable variables
  - from Control.Concurrent.MVar
- Supports synchronized communication w/

```
newEmptyMVar :: IO (MVar a)

newMVar :: a -> IO (MVar a)

takeMVar :: MVar a -> IO a

putMVar :: MVar a -> a -> IO ()

readMVar :: MVar a -> IO a

tryTakeMVar :: MVar a ->
                    IO (Maybe a)

tryPutMVar :: MVar a -> a ->
                    IO Bool

isEmptyMVar :: MVar a ->
                    IO Bool
```

See MVarExample.hs

MVar

- Blocking: Holds only one value
  - Can't take when empty, can't put when full
  - takeMVar destructs value
  - readMVar non-destructive
  - "try" variants may fail, returning false or None

Running Parallel Progs

- Compile with:
  - ghc -threaded -O2 --make parallelProg.hs
- Run with:
  - ./parallelProg +RTS -Nx -RTS
    - where x is number of processors available
    - add -s to get more info on runtime
    - *RTS stands for RunTimeSystem*

Semi-Explicit Parallelism

- Control.Parallel module contains:
  - par :: a -> b -> b -- *usually written as infix `par`*
  - pseq :: a -> b -> b -- *also as `pseq`*
- par indicates to run-time that may be advantageous to evaluate its first argument in parallel w/second, but returns only second
  - Creates a *spark* to compute first argument, but might not compute on separate thread *(or at all!)*

Sorting Examples

```
--Quicksort
sort :: (Ord a) = [a] - [a]
sort (x:xs) = lesser ++ x:greater
   where lesser  = sort [y | y <- xs, y <  x]
         greater = sort [y | y <- xs, y >= x]
sort _ = []
```

Using par

```
hopeSort (x:xs) = greater `par` (lesser ++ x:greater))
   where lesser    = hopeSort [y | y <- xs, y <  x]
         greater  = hopeSort [y | y <- xs, y = x]
hopeSort _       = []
```

- won't help if ++ computed right to left!
- Can't count on order, even if reverse
- greater only computed to head normal form!

Forcing computations

- pseq :: a -> b -> b -- evaluates a then returns b
  - Forces sequential evaluation in same thread

```
force :: [a] -> ()
force xs = go xs `pseq` ()
   where go (_:xs) = go xs
         go [] = 1
```

- Forces full evaluation of list to get answer of 1, which is then thrown away

Using pseq

- Combine pseq and force:

```
parSort :: (Ord a) = [a] - [a]
parSort (x:xs)    = force greater `par` (force lesser `pseq`
                                (lesser ++ x:greater))
   where lesser   = parSort [y | y <- xs, y <  x]
         greater = parSort [y | y <- xs, y = x]
parSort _         = []
```

Can do Better

- Too many "sparks" created, overwhelms system
- Have cut-off on # layers

```
parSort2 :: (Ord a) => Int -> [a] -> [a]
parSort2 d list@(x:xs)
  | d <= 0     = sort list
  | otherwise = force greater `par` (force lesser `pseq`
                              (lesser ++ x:greater))
      where lesser     = parSort2 d' [y | y <- xs, y <  x]
            greater    = parSort2 d' [y | y <- xs, y = x]
            d' = d - 1
parSort2 _ _            = []
```

Performance

- Get about 25% speedup on 4 cores
  - Garbage collection takes up lot of time on this example!

Software Transactional Memory

- Provides atomic memory operations in Control.Concurrent.STM
  - data STM a -- A monad supporting atomic memory transactions
  - atomically :: STM a -> IO a -- Perform a series of STM actions atomically
  - retry :: STM a -- Retry current transaction from the beginning
  - orElse :: STM a -> STM a -> STM a -- Compose two transactions

TVar

- TVars can only be accessed in atomic blocks
  - data TVar a -- Shared memory locations that support atomic memory operations
  - newTVar :: a -> STM (TVar a)
    -- Create a new TVar with an initial value
  - readTVar :: TVar a -> STM a
    -- Return the current value stored in a TVar
  - writeTVar :: TVar a -> a -> STM ()
    -- Write the supplied value into a TVar

Examples

```
type Account = TVar Int
withdraw :: Account -> Int -> STM ()
withdraw acc amount
      = do { bal <- readTVar acc
           ; writeTVar acc (bal - amount) }

deposit :: Account -> Int -> STM ()
deposit acc amount = withdraw acc (- amount)

transfer from to amount
      = atomically (do { deposit to amount
                       ; withdraw from amount })
```

- If interfere, then will automatically retry

Buffer with STM

```
type Buffer a = TVar [a]

newBuffer :: IO (Buffer a)
newBuffer = new TVar IO []

put :: Buffer a -> a -> STM ()
put buffer item = do ls <- readTVar buffer
                     writeTVar buffer (ls ++ [item])

get :: Buffer a -> STM a
get buffer = do ls <- readTVar buffer
            case ls of
              [] -> retry
              (item:rest) -> do writeTVar buffer rest
                                return item
```

Unfortunately...

- In practice seems to have very high overhead
- Indications that 4 to 8 cores required just to match sequential behavior
  - even in the face of no conflicts

Comparing Mechanisms

- Shared memory concurrency
  - Semaphores & locks very low level.
  - Monitors are passive regions encapsulating resources to be shared (mutual exclusion). Cooperation enforced by wait and signal statements.
  - STM would be nice, but seems to have high overhead
- For best results
  - Maximize number of variables accessible by only a single thread
  - Use immutable values wherever possible
  - Use locks or higher-level constructs to avoid data races for all other variables.

Comparing Mechanisms

- Distributed Systems
  - Everything active in Ada tasks (resources and processes) and in Scala actors
  - Monitors and processes can easily be structured as Ada tasks and vice-versa.
  - CML primitives support synchronous and asynchronous communications.
- Problems
  - Must worry about mailboxes filling w/asynchronous message passing.
  - Data in messages must be copied (OK if immutable)

Language Design

Language Design Guidelines

- From Niklaus Wirth
  - designer of Pascal, Modula-2, Oberon
- Writeability
  - Simplicity (& hence easy to master)
  - Uniformity (similar syntax ⇒ similar semantics)
  - Expressiveness
  - Orthogonality & generality
  - Clear, unambiguous syntactic and semantic description

Language Design Guidelines

- Readability
  - Self-documenting
  - Lexical & Syntactic Conventions
  - Not synonymous with wordiness
  - Importance depends on number of programmers

Language Design Guidelines

- Reliability
  - Static checks (including separate compilation)
  - Clear semantics supporting verification
  - Simplicity of compiler implementation (avoid errors)
- Performance
  - Fast translation
  - Efficient object code
  - Machine independence

Wisdom from Hoare

- If interested in features
  - work on 1 at a time in familiar context
  - make sure solves problems without creating new ones
  - Carefully specify semantics
  - Use in lots of examples

Wisdom from Hoare

- If interested in languages
  - know lots of alternatives
  - be especially wary of new features
  - be ready to make modifications to solve minor probs
  - Know intended applications, size & complexity
  - Implement on several machines, write manuals & texts
  - Be prepared to sell it to customers
  - Avoid untried ideas -- consolidation not innovation
  - Make design group as small as possible
  - Simplicity is the key -- avoid complexity

Hoare Advice on Language Design

- Base programming language on
  - Minimum number of independent concepts combined in uniform manner
  - Comprehensive definition mechanism to provide breadth
  - Small core language on which extensions are based
  - Syntax chosen for its readability

Why PLs?

- Deeper understanding of principal features of programming languages
- Explore design space of language features
- Different ways of thinking about programming
- Languages change regularly over time
  - Evaluate suitability for intended purpose
  - Understand choices in design space
- Implementation issues & efficiency

Topics in Recent PL Meetings

- Fixing/Replacing Javascript (types?)
- Gradual types
- Providing security (esp for mobile devices)
- New languages: Go, Dart, Rust, ...
- Concurrency

Class Topics

- Syntax (formal) and semantics (informal and formal) of programming language concepts.
  - Structure of compilers / interpreters.
  - Binding time.
  - Variables: static vs. dynamic scope, lifetime, l-values vs. r-values.
- Run-time structure of programming languages.
  - Allocation of storage at run-time: stack & heap.
  - Parameter passing mechanisms.
  - Storage reclamation - explicit & automatic

Class Topics

- Lambda calculus & functional languages
- OOLs
  - Subtype vs. inheritance (mixins, too)
  - implementation
- Types in programming languages.
  - Available types and their representation.
  - Issues in type-checking & type-inference.
  - Static vs. dynamic type-checking.
  - Problems with pointers.

Class Topics

- Abstract data types
  - Information hiding, encapsulation
  - Modules
- Control structures
  - iterators, exception handling, and continuations.
- Polymorphism - implicit and explicit.
- Concurrency & Parallelism
  - Shared memory, semaphores, locks, monitor
  - Distributed systems, message passing, actors
    - Synchronous vs asynchronous