A monad for deterministic parallelism

Simon Marlow (MSR)
Ryan Newton (Intel)
Parallel programming models

- Deterministic
  - Implicit: FDIP
  - Explicit: par/pseq

- Non-deterministic
  - Concurrent Haskell

Strategies
The Par Monad

Par is a monad for parallel computation

Parallel computations are pure (and hence deterministic)

forking is explicit

results are communicated through IVars

data Par
instance Monad Par

runPar :: Par a -> a

fork :: Par () -> Par ()

data IVar
new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()
Highlights...

- Implemented as a Haskell library
  - almost all the code is in this talk
  - Including a work-stealing scheduler
  - easy to hack on the implementation

- Good performance
  - beats Strategies on some benchmarks
  - but more overhead for very fine-grained stuff
  - programmer has more control

- More explicit and less error-prone than Strategies
  - easier to teach?
Par expresses dynamic dataflow
Examples

• Par can express regular parallelism, like \texttt{parMap}. First expand our vocabulary a bit:

\begin{verbatim}
spawn :: Par a -> Par (IVar a)
spawn p = do r <- new
    fork $ p >>= put r
    return r
\end{verbatim}

• now define \texttt{parMap}:

\begin{verbatim}
parMap :: NFData b => (a -> b) -> [a] -> Par [b]
parMap f xs =
    mapM (spawn . return . f) xs >>= mapM get
\end{verbatim}
Examples

• Divide and conquer parallelism:

```haskell
parfib :: Int -> Int -> Par Int
parfib n
  | n <= 2    = return 1
  | otherwise = do
    x <- spawn $ parfib (n-1)
    y <- spawn $ parfib (n-2)
    x' <- get x
    y' <- get y
    return (x' + y')
```

• In practice you want to use the sequential version when the grain size gets too small
Dataflow

• Consider typechecking a set of (non-recursive) bindings:

  \[
  \begin{align*}
  f &= \ldots \\
  g &= \ldots f \ldots \\
  h &= \ldots f \ldots \\
  j &= \ldots g \ldots h \ldots
  \end{align*}
  \]

• treat this as a dataflow graph:
Dataflow

- No dependency analysis required!
- We just create all the nodes and edges, and let the scheduler do the work
- Maximum parallelism is extracted

```haskell
do
  ivars <- replicateM (length binders) new
  let env = Map.fromList (zip binders ivars)
  mapM_ (fork . typecheck env) bindings
  types <- mapM_ get ivars
  ...
```
Parallel scan

\[
\begin{align*}
\text{scanL } f \, [\_] &= [0] \\
\text{scanL } f \, xs &= \text{interleave } s \, (\text{zipWith } f \, s \, e) \\
\quad \text{where} \\
\quad (e, o) &= \text{uninterleave } xs \\
\quad s &= \text{scanL } f \, (\text{zipWith } f \, e \, o)
\end{align*}
\]

\[
\begin{align*}
\text{scanP'} f \, [\_] &= \text{do } x \leftarrow \text{new}; \text{put } x \, 0; \text{return } [x] \\
\text{scanP'} f \, xs &= \text{do} \\
\quad s &\leftarrow \text{scanP'} f =\lllus \text{parZipWith'} f \, e \, o \\
\quad \text{interleave } s &\lllus \text{parZipWith'} f \, s \, e \\
\quad \text{where} \\
\quad (e, o) &= \text{uninterleave } xs
\end{align*}
\]

\[
\text{parZipWith'} :: \text{NFData } c \\
\quad \Rightarrow (a \rightarrow b \rightarrow c) \\
\quad \rightarrow [\text{IVar } a] \rightarrow [\text{IVar } b] \rightarrow \text{Par } [\text{IVar } c]
\]
Semantics and determinism

- Multiple **put** to the same IVar is an error ($\perp$)
- **runPar** cannot stop when it has the answer. It must run all “threads” to completion, just in case there is a multiple put.
- deadlocked threads are just garbage collected
- Deterministic:
  - a non-deterministic result could only arise from choice between multiple **puts**, which will always lead to an error
  - if the result is an error, it is always an error
  - c.f. determinism proof for CnC
  - care is required with regular $\perp$s (imprecise exceptions to the rescue)
Implementation

• Starting point: A Poor Man’s Concurrency Monad (Claessen JFP’99)
• PMC was used to *simulate* concurrency in a sequential Haskell implementation. We are using it as a way to implement very lightweight non-preemptive threads, with a parallel scheduler.
• Following PMC, the implementation is divided into two:
  – Par computations produce a lazy Trace
  – A scheduler consumes the Traces, and switches between multiple threads
Traces

• A “thread” produces a lazy stream of operations:

```haskell
data Trace
  = Fork Trace Trace
  | Done
  | forall a . Get (IVar a) (a -> Trace)
  | forall a . Put (IVar a) a Trace
  | forall a . New (IVar a -> Trace)
```
The Par monad

- Par is a CPS monad:

```haskell
newtype Par a = Par {
  runCont :: (a -> Trace) -> Trace
}

instance Monad Par where
  return a = Par $ \c -> c a
  m >>= k = Par $ \c -> runCont m $ \a -> runCont (k a) c
```
Operations

fork :: Par () -> Par ()
fork p = Par $ \_c ->
    Fork (runCont p (\_ _ -> Done)) (c ())

new :: Par (IVar a)
new = Par $ \c -> New c

get :: IVar a -> Par a
get v = Par $ \c -> Get v c

put :: NFData a => IVar a -> a -> Par ()
put v a = deepseq a (Par $ \c -> Put v a (c ()))
e.g.

- This code:
  ```plaintext
do
  x <- new
  fork (put x 3)
  r <- get x
  return (r+1)
  ```

- will produce a trace like this:
  ```plaintext
New (\x ->
  Fork (Put x 3 $ Done)
  (Get x (\r ->
    c (r + 1))))
  ```
The scheduler

- First, a sequential scheduler.

```haskell
sched :: SchedState -> Trace -> IO ()

type SchedState = [Trace]
```

The work pool, “runnable threads”

The currently running thread

Why IO? Because we’re going to extend it to be a parallel scheduler in a moment.
newtype IVar a = IVar (IORef (IVarContents a))

data IVarContents a = Full a | Blocked [a -> Trace]
Fork and Done

\[
\text{sched state Done} = \text{reschedule state}
\]

\[
\text{reschedule} :: \text{SchedState} \to \text{IO} ()
\]
\[
\text{reschedule} [] = \text{return} ()
\]
\[
\text{reschedule} (t:ts) = \text{sched ts t}
\]

\[
\text{sched state (Fork child parent)} = 
\text{sched (child:state) parent}
\]
New and Get

sched state (New f) = do
  r <- newIORef (Blocked [])
  sched state (f (IVar r))

sched state (Get (IVar v) c) = do
  e <- readIORef v
  case e of
    Full a -> sched state (c a)
    Blocked cs -> do
      writeIORef v (Blocked (c:cs))
      reschedule state
sched state (Put (IVar v) a t) = do
    cs <- modifyIORef v $ \e -> case e of
        case e of
            Full _        -> error "multiple put"
            Blocked cs   -> (Full a, cs)
    let state' = map ($ a) cs ++ state
    sched state' t

modifyIORef :: IORef a -> (a -> (a,b)) -> IO b

Wake up all the blocked threads, add them to the work pool
Finally... runPar

runPar :: Par a -> a
runPar x = unsafePerformIO $ do
  rref <- newIORef (Blocked [])
  sched [] $ runCont (x >>= put_ (IVar rref))
    (const Done)
  r <- readIORef rref
  case r of
    Full a -> return a
    _      -> error "no result"

rref is an IVar to hold the return value
the “main thread” stores the result in rref
if the result is empty, the main thread must have deadlocked

• that’s the complete sequential scheduler
A real parallel scheduler

- We will create one scheduler thread per core
- Each scheduler has a local work pool
  - when a scheduler runs out of work, it tries to steal from the other work pools
- The new state:

```haskell
data SchedState = SchedState
  { no :: Int,
    workpool :: IORef [Trace],
    idle :: IORef [MVar Bool],
    scheds :: [SchedState]
  }
```
New/Get/Put

• New is the same
• Mechanical changes to Get/Put:
  – use `atomicModifyIORef` to operate on IVars
  – use `atomicModifyIORef` to modify the work pool (now an IORef [Trace], was previously [Trace]).
reschedule :: SchedState -> IO ()
reschedule state@SchedState{ workpool } = do
  e <- atomicModifyIORef workpool $ \ts ->
    case ts of
      [] -> ([], Nothing)
      (t:ts') -> (ts', Just t)
  case e of
    Just t -> sched state t
    Nothing -> steal state

Here’s where we go stealing
stealing

```haskell
steal :: SchedState -> IO ()
steal state@SchedState { scheds, no=me } = go scheds
  where
    go (x:xs)
      | no x == me = go xs
      | otherwise  = do
        r <- atomicModifyIORef (workpool x) $ \ ts ->
          case ts of
            [] -> ([], Nothing)
            (x:xs) -> (xs, Just x)
        case r of
          Just t -> sched state t
          Nothing -> go xs
    go [] = do
      -- failed to steal anything; add ourself to the
      -- idle queue and wait to be woken up
```
runPar :: Par a -> a
runPar x = unsafePerformIO $ do
  let states = ...
  main_cpu <- getCurrentCPU
  m <- newEmptyMVar
  forM_ (zip [0..] states) $ \(cpu,state) ->
    forkOnIO cpu $ 
    if (cpu /= main_cpu)
      then reschedule state
      else do
        rref <- newIORef Empty
        sched state $
        runCont (x >>= put_ (IVar rref))
        (const Done)
        readIORef rref >>= putMVar m
  r <- takeMVar m
  case r of Full a -> return a
           _    -> error "no result"

The “main thread” runs on the current CPU, all other CPUs run workers

An MVar communicates the result back to the caller of runPar
Results

<table>
<thead>
<tr>
<th>Cores</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>99%</td>
<td></td>
</tr>
<tr>
<td>95%</td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing speedup vs cores for blackscholes, minimax, and mandel.](image)
Optimisation possibilities

- Unoptimised it performs rather well
- The overhead of the monad and scheduler is visible when running parFib
- Deforest away the Trace
  - Mechanical; just define
    ```haskell
    type Trace = SchedState -> IO ()
    ```
  - and each constructor in the Trace type is replaced by a function, whose implementation is the appropriate case in `sched`
  - this should give good results but currently doesn’t
More optimisation possibilities

• Use real lock-free work-stealing queues
  – We have these in the RTS, used by Strategies
  – could be exposed via primitives and used in Par

• Give Haskell more control over scheduling?
Extending with CnC functionality

- Generalise IVars to mappings

```haskell
data ItemSet k v
newItemSet :: Par (ItemSet k v)
getItem :: Ord k => ItemSet k v -> k -> Par v
putItem :: Ord k => ItemSet k v -> k -> v -> Par ()
```

- e.g. in the parallel typechecking example earlier, no need to pre-populate the environment

```haskell
do
  env <- newItemSet
  mapM_ (fork . typecheck env) bindings
  types <- mapM_ (getItem env) binders
```

get blocks if the ItemSet does not have a value for that key yet
Could Par be a monad transformer?

- No.
Modularity

• Key property of Strategies is modularity
  \[
  \text{parMap } f \text{ xs } = \text{map } f \text{ xs `using` parList} \text{ rwhnf}
  \]

• Relies on lazy evaluation
  – fragile
  – not always convenient to build a lazy data structure

• Par takes a different approach to modularity:
  – the Par monad is for \textit{coordination} only
  – the application code is written separately as pure Haskell functions
  – The “parallelism guru” writes the coordination code
  – \textbf{Par} performance is not critical, as long as the grain size is not too small
Drawbacks

• Nesting isn’t handled well. Each runPar creates a new gang of threads.
• GHC doesn’t optimise the CPS very well (yet).
Related work

• **Evaluation Strategies**
  – Par is more explicit; no reliance on lazy evaluation (programmer has more control)
  – Par is less modular (though modularity can be achieved in a different way)
  – Par requires no special RTS support, implemented as a library

• **Concurrent Haskell**
  – but Par is deterministic

• **CnC**
  – Haskell CnC is the forerunner to Par
  – Par is dynamic and does not have map-based synchronisation variables (but they could be added)

• **Cilk**
  – but Par has async dataflow

• **pH**
  – Par has explicit forking, and does not modify Haskell