The Eval Monad and Strategies

Stephen A. Edwards

Columbia University

Fall 2023
The Eval Monad: rpar and rseq
Speeding up a Sudoku Solver
  Static Partitioning
  Dynamic Partitioning
  Amdahl’s Law
Control.DeepSeq: Reducing to Normal Form
  rnf for a Custom Data Type
Evaluation Strategies
  Basic Strategies
  Composing Strategies
  Evaluating Lists in Parallel
Example: The K-Means Problem
  Generating a Data Set
  Parallelizing K-Means
  Performance of Parallel K-Means
Garbage Collected Sparks
Parallelizing Lazy Streams
The Eval Monad: An Alternative to \texttt{seq} and \texttt{par}

When in doubt, create a Monad

```haskell
module Control.Parallel.Strategies where

data Eval a = ...

instance Monad Eval where ...

runEval :: Eval a -> a  -- Get the result

rpar :: a -> Eval a    -- Spark evaluation
rseq :: a -> Eval a    -- Wait for evaluation to WHNF
```
runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  return (a, b)

Start parallel evaluation of f x and f y
Return immediately
Start parallel evaluation of \( f \ x \) and \( f \ y \)

Wait for \( f \ y \) to finish, then return

*Do we really know \( f \ y \) is faster?*
```haskell
runEval $ do
  a <- rpar (f x)
  b <- rseq (f y)
  rseq a
  return (a, b)
```

Equivalent and symmetrical:

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

Start parallel evaluation of \( f \, x \) and \( f \, y \)
Wait for both to finish, then return
Marlow’s Sudoku Solver Example (single-threaded)

```
$ wget https://github.com/simonmar/parconc-examples/archive/master.tar.gz
$ tar --strip-components=1 -zxf master.tar.gz \
   parconc-examples-master/{Sudoku.hs,sudoku17.1000.txt}

import Sudoku ; import Control.Exception
import System.Environment ; import Data.Maybe

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

```
$ stack ghc -- -O2 sudoku1.hs -rtsopts
$ ./sudoku1 sudoku17.1000.txt +RTS -s
```
import Sudoku
import Data.Maybe; import System.Environment
import Control.Parallel.Strategies; import Control.DeepSeq

main :: IO ()
main = do puzzles <- getArgs >>= \[f] -> lines <$> readFile f

  let (as,bs) = splitAt (length puzzles `div` 2) puzzles

  solutions = runEval $ do
    as' <- rpar (force (map solve as))  -- To normal
    bs' <- rpar (force (map solve bs))  -- form
    _ <- rseq as'
    _ <- rseq bs'
    return (as' ++ bs')

  print (length (filter isJust solutions))
The Problem with Static Partitioning

```
$ stack install parallel
$ stack ghc -- -O2 -Wall sudoku2.hs -threaded -rtsopts -eventlog
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -s -ls
$ threadscope sudoku2.eventlog
```

Speedup on two cores: $\frac{1.37}{0.891} = 1.53 \times$
import Sudoku
import System.Environment ; import Data.Maybe
import Control.Parallel.Strategies hiding (parMap)

parMap :: (a -> b) -> [a] -> Eval [b]
parMap _ [] = return []
parMap f (a:as) = do b <- rpar (f a) -- Spark evaluation of f a
                    bs <- parMap f as -- Recurse
                   return (b:bs)

main :: IO ()
main = do puzzles <- getArgs >>= \[f] -> lines <$> readFile f

       let solutions = runEval (parMap solve puzzles)

       print (length (filter isJust solutions))
The Advantage of Dynamic Partitioning

$\text{stack ghc -- -O2 -Wall sudoku3.hs -threaded -rtsopts -eventlog}$

$\text{./sudoku3 sudoku17.1000.txt +RTS -N2 -s -ls}$

$\text{threadscope sudoku3.eventlog}$

Speedup on two cores: $1.37/0.751 = 1.82 \times$ versus $1.53 \times$ with static partitioning
Amdahl’s Law: Why Parallelism is Difficult

Validity of the single processor approach to achieving large scale computing capabilities

by DR. GENE M. AMDAHL
International Business Machines Corporation
Sunnyvale, California

AFIPS Joint Computer Conference, 1967

\[ S = \frac{1}{(1 - P) + P/N} \]

\( \lim_{N \to \infty} S \) is the depressing part:

If \( P = 0.5 \), \( S \to 2 \) as \( N \to \infty \)

If \( P = 0.95 \), \( S \to 20 \) as \( N \to \infty \)

\( P \) is the parallel fraction of the task

\( N \) is the degree of parallelism

\( S \) is the speedup
DeepSeq and Normal Form

Weak Head Normal Form = Top is data constructor or lambda, not application

Normal Form = Data constructors or lambdas all the way down

In Control.Deepseq,

class NFData a where
  rnf :: a -> ()  -- "Reduce to Normal Form"
  rnf a = a `seq` ()  -- Default, e.g., for numbers

depseq :: NFData a => a -> b -> b
depseq a b = rnf a `seq` b  -- Evaluate a to NF; return b

force :: NFData a => a -> a
force a = a `deepseq` a  -- Evaluate a to NF; return it

($!!) :: NFData a => (a -> b) -> a -> b
f $!! x = x `deepseq` f x  -- evaluate x then f x

Control.Deepseq has instances of NFData for standard types (numbers, characters, lists, tuples)
Normal Form vs. Weak Head Normal Form

Prelude> import Control.DeepSeq
Prelude Control.DeepSeq> let x = [1..10] :: [Int]
Prelude Control.DeepSeq> :sprint x
x = _
Prelude Control.DeepSeq> :sprint x
x = _
Prelude Control.DeepSeq> x `seq` ()
()}
Prelude Control.DeepSeq> (last $ take 3 x) `seq` ()
()}
Prelude Control.DeepSeq> :sprint x
x = 1 : _
Prelude Control.DeepSeq> :sprint x
x = 1 : 2 : 3 : _
Prelude Control.DeepSeq> x `deepseq` ()
()}
Prelude Control.DeepSeq> :sprint x
x = [1,2,3,4,5,6,7,8,9,10]
Roll-your-own NFData

```haskell
import Control.DeepSeq

data Tree a = Empty | Branch (Tree a) a (Tree a)

instance NFData a => NFData (Tree a) where
  rnf Empty = ()
  rnf (Branch l a r) = rnf l `seq` rnf a `seq` rnf r
```

```haskell
*Main> let singleton x = Branch Empty x Empty
*Main> let x = Branch (singleton 'a') 'b' (singleton 'c')
*Main> :sprint x
x = Branch _ 'b' _
*Main> x `seq` ()
()
*Main> :sprint x
x = Branch _ 'b' _
*Main> rnf x
()
*Main> :sprint x
x = Branch (Branch Empty 'a' Empty) 'b' (Branch Empty 'c' Empty)
```
“Strategies are a means for modularizing parallel code by separating the algorithm from the parallelism. Sometimes they require you to rewrite your algorithm, but once you do so, you will be able to parallelize it in different ways just by substituting a new Strategy.” —Simon Marlow

1. Build a lazy data structure representing the computation
2. Apply a Strategy that traverses the computation applying \texttt{rpar} and \texttt{rseq}

A Strategy: an identity function in the \texttt{Eval} monad (Control.Parallel.Strategies)

\begin{verbatim}
  type Strategy = a -> Eval a
\end{verbatim}
### An Example: A Parallel Strategy for Pairs

<table>
<thead>
<tr>
<th>rpar :: Strategy a</th>
<th>-- Spark evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>runEval :: Eval a -&gt; a</td>
<td>-- Evaluate; return value</td>
</tr>
<tr>
<td>parPair :: Strategy (a,b)</td>
<td>-- Simple parallel strategy for pairs</td>
</tr>
<tr>
<td>parPair (a,b) = do</td>
<td></td>
</tr>
<tr>
<td>a' &lt;- rpar a</td>
<td>-- Spark parallel evaluation of</td>
</tr>
<tr>
<td>b' &lt;- rpar b</td>
<td>-- a and b to WHNF</td>
</tr>
<tr>
<td>return (a',b')</td>
<td></td>
</tr>
</tbody>
</table>

```haskell```
runEval (parPair (fib 35, fib 36))
```

More elegantly,

```haskell```
using :: a -> Strategy a -> a | -- In Control.Parallel.Strategies
x `using` s = runEval (s x) | -- Apply s, return x

(fib 35, fib 36) `using` parPair
```
Basic Strategies

In Control.Parallel.Strategies:

\[ r0 :: \text{Strategy} \ a \]
\[ r0 \ x = \text{return} \ x \quad -- \text{Do not evaluate} \ x \]

\[ \text{rseq} :: \text{Strategy} \ a \quad -- \text{Evaluate to WHNF; wait for completion} \]

\[ \text{rdeepseq} :: \text{NFData} \ a \Rightarrow \text{Strategy} \ a \quad -- \text{Fully evaluate then proceed} \]
\[ \text{rdeepseq} \ x = \text{rseq} \ (\text{rnf} \ x) \gg \text{return} \ x \]

\[ \text{rpar} :: \text{Strategy} \ a \quad -- \text{Spark evaluation (in parallel) to WHNF} \]

\[ \text{rparWith} :: \text{Strategy} \ a \Rightarrow \text{Strategy} \ a \]
\[ \text{rparWith} \ s \ x \quad -- \text{Spark evaluation of} \ x \text{ ‘using’} \ s \]
Building Strategies from Strategies

A skeleton for expressing strategies for evaluating tuples:

-- In Control.Parallel.Strategies,

```
 evalTuple2 :: Strategy a -> Strategy b -> Strategy (a, b)
evalTuple2 sa sb (a, b) = do a' <- sa a
                            b' <- sb b
                            return (a', b')
```

```
parPair :: Strategy (a, b)
parPair = evalTuple2 rpar rpar -- Spark elements’ evaluation to WHNF
```

What if we wanted to fully evaluate the two elements in parallel?

```
parPair :: Strategy a -> Strategy b -> Strategy (a, b)
parPair sa sb = evalTuple2 (rparWith sa) (rparWith sb)
```

```
parPair rdeepseq rdeepseq (fib 25, fib 26)
```
parPair rdeepseq rdeepseq (a, b)

“Spark two parallel threads that fully evaluate a and b to normal form”

A cartoon of how this works:

parPair rdeepseq rdeepseq (a, b)
    = evalTuple2 (rparWith rdeepseq) (rparWith rdeepseq) (a, b)

    = do a' <- (rparWith rdeepseq) a
        b' <- (rparWith rdeepseq) b
        return (a', b')

    = do a' <- rpar (a `using` \x -> rseq (rnf x) >> return x)
        b' <- rpar (b `using` \x -> rseq (rnf x) >> return x)
        return (a', b')
In Control.Parallel.Strategies,

\[
\text{evalList} :: \text{Strategy } a \rightarrow \text{Strategy } [a] \quad -- \text{Apply a strategy}
\]

\[
\text{evalList } _\cdot [ ] = \text{return } [ ] \quad -- \text{to each list element}
\]

\[
\text{evalList } s \ (x:xs) = \text{do } x' \leftarrow s \ x
\]
\[
\quad \text{xs'} \leftarrow \text{evalList } s \ xs
\]
\[
\quad \text{return } (x':xs')
\]

\[
\text{parList} :: \text{Strategy } a \rightarrow \text{Strategy } [a] \quad -- \text{Evaluate each list element}
\]

\[
\text{parList } s = \text{evalList } (\text{rparWith } s) \quad -- \text{in parallel with strategy}
\]

Combining these to evaluate all list elements to WHNF in parallel:

\[
\text{parMap} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{parMap } f \ xs = \text{map } f \ xs \ `\text{using}` \ \text{parList } \text{rseq}
\]
Simpler Parallel Sudoku

```haskell
import Sudoku(solve)
import System.Environment(getArgs)
import Data.Maybe(isJust)
import Control.Parallel.Strategies(using, parList, rseq)

main :: IO ()
main = do [fname] <- getArgs
    puzzles <- lines <$> readFile fname

    let solutions = map solve puzzles `using` parList rseq

    print $ length $ filter isJust solutions
```

Note that `rseq` only evaluates to WHNF, but that suffices for Sudoku.

About the same performance as the “parMap” version presented earlier.
Example: The K-Means Problem

Lloyd’s (approximation) algorithm

Give a number of clusters $k$,

1. Guess a center for each cluster
2. Group points by closest centerpoint
3. Calculate the centroid (average) of each group
4. Repeat steps 3–4 until satisfied
Example: The K-Means Problem

```bash
$ wget https://github.com/simonmar/parconc-examples/archive/master.tar.gz
$ tar --strip-components=1 -zxf master.tar.gz \
  parconc-examples-master/kmeans
```

2D points (to simplify visualization) and clusters, in KMeansCore.hs,

```haskell
data Point = Point !Double !Double  -- ! disables laziness

zeroPoint :: Point
zeroPoint = Point 0 0

sqDistance :: Point -> Point -> Double  -- Distance squared for speed
sqDistance (Point x1 y1) (Point x2 y2) = ((x1-x2)^2) + ((y1-y2)^2)

data Cluster = Cluster { clId :: Int  -- number of this cluster
                      , clCent :: Point  -- centroid of this cluster
                      }
```
Example: The K-Means Problem

For computing the centroids (average of all points in a cluster), in kmeans.hs,

```haskell
data PointSum = PointSum !Int !Double !Double

addToPointSum :: PointSum -> Point -> PointSum
addToPointSum (PointSum count xs ys) (Point x y)
    = PointSum (count+1) (xs + x) (ys + y)

pointSumToCluster :: Int -> PointSum -> Cluster
pointSumToCluster i (PointSum count xs ys) = Cluster {
  clId   = i
  , clCent = Point (xs / fromIntegral count)
                 (ys / fromIntegral count)
}
```
1. Accumulate Points in PointSums for Nearest Centroid

assign :: Int -> [Cluster] -> [Point] -> Vector PointSum
assign nclusters clusters points = Vector.create $ do
  vec <- MVector.replicate nclusters (PointSum 0 0 0)

  let addpoint p = do
    let c = nearest p ; cid = clId c
    ps <- MVector.read vec cid
    MVector.write vec cid $! addToPointSum ps p

    mapM_ addpoint points
  return vec

where
  nearest p = fst $ minimumBy (compare `on` snd)
    [ (c, sqDistance (clCent c) p) | c <- clusters ]

Vectors are Haskell’s fixed-length, random-access arrays that are “mutable” in the right monad. See Data.Vector and Data.Vector.Mutable
2. Create New Clusters from PointSums

```haskell
makeNewClusters :: Vector PointSum -> [Cluster]
makeNewClusters vec =
    [ pointSumToCluster i ps |
      (i,ps@(PointSum count _ _)) <- zip [0..] (Vector.toList vec), count > 0
    ]
```

One step of the algorithm: group by nearest centroid; calculate new centroids

```haskell
step :: Int -> [Cluster] -> [Point] -> [Cluster]
step nclusters clusters points
    = makeNewClusters (assign nclusters clusters points)
```
The Sequential Loop: step until converged or give up

\[
k\text{means}\_\text{seq} :: \textbf{Int} \rightarrow \textbf{[Point]} \rightarrow \textbf{[Cluster]} \rightarrow \textbf{IO} \rightarrow \textbf{[Cluster]} \\
k\text{means}\_\text{seq} n\text{clusters} \text{ points} \text{ clusters} = \\
\textbf{let} \> \text{loop} :: \textbf{Int} \rightarrow \textbf{[Cluster]} \rightarrow \textbf{IO} \rightarrow \textbf{[Cluster]} \\
\text{loop} n \text{ clusters} \mid n > \text{tooMany} = \textbf{do} \\
\> \text{putStrLn} \> "giving up." \\
\> \textbf{return} \> \text{clusters} \\
\text{loop} n \text{ clusters} = \textbf{do} \\
\> \text{printf} \> "iteration \%d
" \> n \\
\> \text{putStr} \> \text{unlines} \> \text{map} \> \text{show} \> \text{clusters} \\
\textbf{let} \> \text{clusters}' = \textbf{step} \> n\text{clusters} \text{ clusters} \text{ points} \\
\> \text{if} \> \text{clusters}' \> \text{==} \> \text{clusters} \\
\> \> \textbf{then} \> \textbf{return} \> \text{clusters} \\
\> \> \textbf{else} \> \text{loop} \> (n+1) \> \text{clusters}' \\
\textbf{in} \> \text{loop} \> 0 \> \text{clusters} \\
\]

\text{tooMany} = 80
Generating a Data Set

```sh
$ cabal install normaldistribution
$ ghc -O2 GenSamples.hs
$ ./GenSamples 5 50000 100000 1010
$ ls -l points.bin
-rw-rw-r-- 1 sedwards sedwards 16M Nov 23 14:58 points.bin
$ gnuplot -e 'set terminal png;set nokey;plot "points"' > points.png
```
Compiling and Running K-Means

```bash
$ stack install monad-par
$ cd kmeans
$ stack ghc -- -O2 -threaded -rtsopts -eventlog kmeans.hs
```

Run it in sequential mode:

```bash
$ ./kmeans seq
...
iteration 20
Cluster {clId = 0, clCent = Point -5.84359465 -5.46502314}
Cluster {clId = 1, clCent = Point 8.316354592 -8.33043084}
Cluster {clId = 2, clCent = Point -9.06455081 7.561852464}
Cluster {clId = 3, clCent = Point 9.243597731 6.138576051}
Cluster {clId = 4, clCent = Point -3.62170911 -1.82458124}
Total time: 0.73
```
Parallelizing K-Means

Computing nearest center for each point is the main operation to parallelize. This is a fold with an associative accumulation function `addToPointSum`.

Too many points and not enough work per point for per-point parallelism; overhead would dominate. Better to split work into coarser chunks.

```haskell
split :: Int -> [a] -> [[a]]  -- Divide into numChunks chunks
split numChunks xs = chunk (length xs `quot` numChunks) xs

chunk :: Int -> [a] -> [[a]]  -- Split into n-point chunks
chunk _ [] = []
chunk n xs = let (as,bs) = splitAt n xs in as : chunk n bs

addPointSums :: PointSum -> PointSum -> PointSum  -- Accumulate PointSums
addPointSums (PointSum c1 x1 y1) (PointSum c2 x2 y2)
  = PointSum (c1+c2) (x1+x2) (y1+y2)

combine :: Vector PointSum -> Vector PointSum -> Vector PointSum
combine = Vector.zipWith addPointSums  -- Accumulate vectors
```
Code for a Parallel step

Analyze the chunks in parallel; merge; and make new clusters:

```haskell
parSteps_strat :: Int -> [Cluster] -> [[Point]] -> [Cluster]
parSteps_strat nclusters clusters pointss
    = makeNewClusters $ foldr1 combine $ (map (assign nclusters clusters) pointss)
    `using` parList rseq
```

--- Merge the results from each chunk
--- Analyze chunks
--- in parallel
The Parallel Loop: Divide into chunks; apply parSteps_strat

\[ \text{kmeans\_strat :: Int -> Int -> [Point] -> [Cluster] -> IO [Cluster]} \]
\[ \text{kmeans\_strat numChunks nclusters points clusters =} \]
\[ \text{let chunks = split numChunks points} \quad -- \text{One big change} \]
\[ \text{loop :: Int -> [Cluster] -> IO [Cluster]} \]
\[ \text{loop n clusters | n > tooMany = do} \]
\[ \text{printf "giving up."} \]
\[ \text{return clusters} \]
\[ \text{loop n clusters = do} \]
\[ \text{printf "iteration \%d\n" n} \]
\[ \text{putStr (unlines (map show clusters))} \]
\[ \text{let clusters' = parSteps\_strat nclusters clusters chunks} \]
\[ \text{if clusters' == clusters} \]
\[ \text{then return clusters} \]
\[ \text{else loop (n+1) clusters'} \]
\[ \text{in loop 0 clusters} \]
Performance of `kmeans_strat` on 1–8 Cores

```
./kmeans_strat 64 +RTS -N1
./kmeans_strat 64 +RTS -N8
```

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.77</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.40</td>
<td>1.91</td>
</tr>
<tr>
<td>3</td>
<td>0.29</td>
<td>2.70</td>
</tr>
<tr>
<td>4</td>
<td>0.22</td>
<td>3.45</td>
</tr>
<tr>
<td>5</td>
<td>0.23</td>
<td>3.28</td>
</tr>
<tr>
<td>6</td>
<td>0.22</td>
<td>3.45</td>
</tr>
<tr>
<td>7</td>
<td>0.22</td>
<td>3.47</td>
</tr>
<tr>
<td>8</td>
<td>0.25</td>
<td>3.10</td>
</tr>
</tbody>
</table>

Using “Total time” reported by the program; ignores reading point data
Lots of sequential file processing start: not being counted against speedup
Off to a difficult start; iterations are periodic hiccups; big garbage collect
Program suddenly turns completely sequential; darn Amdahl.
Marlow found printing was a major culprit, but removing it didn’t matter on
Iteration: sudden spark creation activity in single HEC pool, then slow conversion. Main thread migrated after 8 iterations. (Under the “Traces” tab)
The Effects of Granularity (N=8)

Time (ms) → unbalanced

overhead dominates →

Chunks
parList/evalList creates a new list, which seems wasteful

\[
\text{parList} :: \text{Strategy} \ a \\
\rightarrow \text{Strategy} \ [a]
\]

\[
\text{parList} \ s = \\
\quad \text{evalList} (\text{rparWith} \ s)
\]

\[
\text{evalList} :: \text{Strategy} \ a \\
\rightarrow \text{Strategy} \ [a]
\]

\[
\text{evalList} \ _ \ [] = \text{return} \ []
\]

\[
\text{evalList} \ s \ (x:xs) = \\
\quad \text{do} \ x' \leftarrow s \ x \\
\quad \quad \text{xss'} \leftarrow \text{evalList} \ s \ xs \\
\quad \text{return} \ (x':xs') \quad \quad \text{-- Cons}
\]

Consider this walk-the-list alternative that “touches” elements with \textit{rparWith}

\[
\text{parList} :: \text{Strategy} \ a \\
\rightarrow \text{Strategy} \ [a]
\]

\[
\text{parList} \ \text{strat} \ xs = \text{do} \ \text{go} \ xs \\
\quad \text{return} \ xs
\]

\[
\text{where}
\]

\[
\text{go} \ [] = \text{return} \ ()
\]

\[
\text{go} \ (x:xs) = \text{do} \ \text{rparWith} \ \text{strat} \ x \\
\quad \text{go} \ xs
\]

Doesn’t work: each spark created by \textit{rparWith} is garbage-collected because it is never used. Critical that the result of \textit{rpar/rparWith} be returned.
Heap Layout of Working parList: New List Inhibits Spark GC

Marlow, fig. 3-8

Array of pointers to sparks

Original list

List elements

Sparks: apply the strategy to evaluate list elements

← New list provides second pointer to each spark

New list
Parallelizing Lazy Streams: RSA Encoder/Decoder from Marlow

```
$ stack ghc -- -O2 -Wall -rtsopts rsa
$ ./rsa encrypt /usr/share/dict/words > /dev/null +RTS -s
  5,089,757,232 bytes allocated in the heap
  3,043,360 bytes copied during GC
  107,888 bytes maximum residency (3 sample(s))
  27,968 bytes maximum slop
  0 MB total memory in use

Total    time  5.740s ( 5.767s elapsed)
```

```
$ ls -sh /usr/share/dict/american-english
920K /usr/share/dict/american-english
```

Dictionary file is about 1 MB, but runtime only uses 107,888 bytes maximum because of Data.ByteString.Lazy.Char8
Parallelizing RSA

Sequential implementation:

```haskell
encrypt :: Integer -> Integer -> ByteString -> ByteString
encrypt n e = B.unlines
  . map (B.pack . show . power e n . code) -- Encrypt
  . chunk (size n) -- Split
```

First try (rsa1.hs): use `parList rdeepseq`

```haskell
encrypt n e = B.unlines
  . withStrategy (parList rdeepseq)
  . map (B.pack . show . power e n . code)
  . chunk (size n)
```

```haskell
withStrategy s e = e `using` s
```
Speedup using `parList rdeepseq`

```
$ stack ghc -- -O2 -Wall -threaded -rtsopts rsa1
$ ./rsa1 encrypt /usr/share/dict/words > /dev/null +RTS -N8 -s
  5,319,033,432 bytes allocated in the heap
  18,619,728 bytes copied during GC
  3,029,464 bytes maximum residency (10 sample(s))
  570,920 bytes maximum slop
       2 MB total memory in use

SPARKS: 9988
       (8254 converted, 1734 overflowed, 0 dud, 0 GC'd, 0 fizzled)

Total time 14.403s ( 2.991s elapsed)
```

Speedup of 1.92 over sequential (rsa.hs) (4.8× itself)

Maximum memory use now 3 MB (cf. 107 KB): `parList` traverses the whole list.
Control.Parallel.Strategies.parBuffer: Regulate number of outstanding sparks

`parBuffer 100` creates 100 outstanding sparks; sparks more once consumed

```
parBuffer :: Int -> Strategy a -> Strategy [a]
```

```
encrypt n e = B.unlines -- rsa2.hs
  withStrategy (parBuffer 100 rdeepseq) -- 100 max
  map (B.pack . show . power e n . code)
  chunk (size n)
```

```
$ ./rsa2 encrypt /usr/share/dict/words > /dev/null +RTS -N8 -s
  506,640 bytes maximum residency (18 sample(s))
  SPARKS: 9988
    (9987 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)
  Total time 12.160s ( 1.641s elapsed)
```

Down to 500 KB residency, 3.5× over sequential, excellent 7.4× self-speedup
HEC5 got the spark creation process
Spark pool remains around 100
Hiccups primarily garbage collection

No creation or pools on other HECs
Gray on graph denotes variance
About 11% overhead
RSA Strategies (parList, parBuffer) Compared

<table>
<thead>
<tr>
<th>Technique</th>
<th>Memory (K)</th>
<th>Sparks</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Converted</td>
<td>Overflowed</td>
<td>Dud</td>
</tr>
<tr>
<td>Sequential</td>
<td>105</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>parList</td>
<td>2958</td>
<td>8254</td>
<td>1734</td>
<td>0</td>
</tr>
<tr>
<td>parBuffer</td>
<td>495</td>
<td>9987</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Both generate the same number of sparks

*parList* forces the entire file to be loaded (memory consumption) and generates all the sparks at the beginning (spark pool overflow).