Summer School on Advances in Programming Languages

Glasgow parallel Haskell

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

Characteristics of Functional Languages

GpH is a conservative extension to the purely-functional, non-strict language Haskell.
Thus, GpH provides all the of the advanced features inherited from Haskell:

- Sophisticated polymorphic type system, with type inference
- Pattern matching
- Higher-order functions
- Data abstraction
- Garbage-collected storage management

Most relevant for parallel execution is *referential transparency*:

> *The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value.* *[Stoy, 1977]*

Consequences of Referential Transparency

Equational reasoning:

- Proofs of correctness are much easier than reasoning about state as in procedural languages.
- Used to *transform* programs, e.g. to transform simple specifications into efficient programs.

Freedom from execution order:

- Meaning of program is not dependent on execution order.
- *Lazy evaluation:* an expression is only evaluated when, and if, it is needed.
- *Parallel/distributed evaluation*. Often there are many expressions that can be evaluated at a time, because we know that the order of evaluation doesn't change the meaning, the sub-expressions can be evaluated in parallel (Wegner 1978)

Elimination of *side effects* (unexpected actions on a global state).

The Challenge of Parallel Programming

Engineering a parallel program entails specifying

- *computation*: a correct, efficient algorithm
  in GpH the semantics of the program is unchanged
- *coordination*: arranging the computations to achieve "good" parallel behaviour.
  in GpH coordination and computation are cleanly separated

Coordination Aspects

Coordinating parallel behaviour entails, *inter alia*:

- partitioning
  - what threads to create
  - how much work should each thread perform
- thread synchronisation
- load management
- communication
- storage management

Specifying full coordination details is a significant burden on the programmer

High Level Parallel Programming

High level parallel programming aims to reduce the programmer's coordination management burden.

This can be achieved through skeletons (*Eden*), through specific execution models (array languages such as *SAC*, dataflow languages such as *Swan*), or parallelising compilers (*pH*).

GpH (Glasgow parallel Haskell) uses a model of *semi-explicit* parallelism: only a few key aspects of coordination need to be specified by the programmer.

High Level Parallel Programming

High level parallel programming aims to reduce the programmer's coordination management burden.

This can be achieved through skeletons (*Eden*), through specific execution models (array languages such as *SAC*, dataflow languages such as *Swan*), or parallelising compilers (*pH*).

GpH (Glasgow parallel Haskell) uses a model of *semi-explicit* parallelism: only a few key aspects of coordination need to be specified by the programmer.

GpH Coordination Primitives

GpH provides parallel composition to *hint* that an expression may usefully be evaluated by a parallel thread.
We say `x` is *"sparked"*: if there is an idle processor a thread may be created to evaluate it.

Evaluation

`x `par` y` $\Rightarrow$ `y`

GpH provides sequential composition to sequence computations and specify how much evaluation a thread should perform. `x` is evaluated to Weak Head Normal Form (WHNF) before returning `y`.

Evaluation

`x `pseq` y` $\Rightarrow$ `y`

GpH Coordination Primitives

GpH provides parallel composition to *hint* that an expression may usefully be evaluated by a parallel thread.
We say `x` is *"sparked"*: if there is an idle processor a thread may be created to evaluate it.

Evaluation

`x `par` y` $\Rightarrow$ `y`

GpH provides sequential composition to sequence computations and specify how much evaluation a thread should perform. `x` is evaluated to Weak Head Normal Form (WHNF) before returning `y`.

Evaluation

`x `pseq` y` $\Rightarrow$ `y`

GpH Coordination Primitives

GpH provides parallel composition to *hint* that an expression may usefully be evaluated by a parallel thread.
We say `x` is *"sparked"*: if there is an idle processor a thread may be created to evaluate it.

Evaluation

`x `par` y` $\Rightarrow$ `y`

GpH provides sequential composition to sequence computations and specify how much evaluation a thread should perform. `x` is evaluated to Weak Head Normal Form (WHNF) before returning `y`.

Evaluation

`x `pseq` y` $\Rightarrow$ `y`

Introducing Parallelism: a GpH Factorial

Factorial is a classic *divide and conquer* algorithm.

Example (Parallel factorial)

```
pfact n = pfact' 1 n

pfact' :: Integer -> Integer -> Integer
pfact' m n
  | m == n     = m
  | otherwise = left `par` right `pseq` (left * right)
      where mid   = (m + n) `div` 2
            left  = pfact' m mid
            right = pfact' (mid+1) n
```

Controlling Evaluation Order

Notice that we must *control evaluation order:* If we wrote the function as follows, then the addition may evaluate `left` on this core/processor before any other has a chance to evaluate it

```
  | otherwise = left `par` (left * right)
```

The `right `pseq`` ensures that `left` and `right` are evaluated before we multiply them.

Controlling Evaluation Degree

In a non strict language we must specify *how much* of a value should be computed.

For example the obvious quicksort produces almost no parallelism because the threads reach WHNF very soon: once the first cons cell of the sublist exists!

Example (Quicksort)

```
quicksortN :: (Ord a) => [a] -> [a]
quicksortN []     = []
quicksortN [x]    = [x]
quicksortN (x:xs) =
  losort `par`
  hisort `par`
  losort ++ (x:hisort)
    where
      losort = quicksortN [y|y <- xs, y < x]
      hisort = quicksortN [y|y <- xs, y >= x]
```

Controlling Evaluation Degree (cont'd)

Forcing the evaluation of the sublists gives the desired behaviour:

Example (Quicksort with forcing functions)

```
forceList :: [a] -> ()
forceList [] = ()
forceList (x:xs) = x `pseq` forceList xs

quicksortF []     = []
quicksortF [x]    = [x]
quicksortF (x:xs) =
  (forceList losort) `par`
  (forceList hisort) `par`
  losort ++ (x:hisort)
    where
      losort = quicksortF [y|y <- xs, y < x]
      hisort = quicksortF [y|y <- xs, y >= x]
```

Problem: we need a different forcing function for each datatype.

GpH Coordination Aspects

To specify parallel coordination in Haskell we must

1. Introduce parallelism
2. Specify Evaluation Order
3. Specify Evaluation Degree

This is much less than most parallel paradigms, e.g. no communication, synchronisation etc.

It's important that we do so without cluttering the program. In many parallel languages, e.g. C with MPI, coordination so dominates the program text that it obscures the computation.

Evaluation Strategies: Separating Computation and Coordination

Evaluation Strategies abstract over `par` and `pseq`,

- raising the level of abstraction, and
- separating coordination and computation concerns

It should be possible to understand the semantics of a function without considering its coordination behaviour.

Evaluation Strategies

An *evaluation strategy* is a function that specifies the coordination required when computing a value of a given type, and preserves the value i.e. it is an identity function.

```
type Strategy a = a -> Eval a

data Eval a = Done a
```

We provide a simple function to extract a value from `Eval`:

```
runEval :: Eval a -> a
runEval (Done a) = a
```

The `return` operator from the `Eval` monad will introduce a value into the monad:

```
return :: a -> Eval a
return x = Done x
```

Evaluation Strategies

An *evaluation strategy* is a function that specifies the coordination required when computing a value of a given type, and preserves the value i.e. it is an identity function.

```
type Strategy a = a -> Eval a

data Eval a = Done a
```

We provide a simple function to extract a value from `Eval`:

```
runEval :: Eval a -> a
runEval (Done a) = a
```

The `return` operator from the `Eval` monad will introduce a value into the monad:

```
return :: a -> Eval a
return x = Done x
```

Evaluation Strategies

An *evaluation strategy* is a function that specifies the coordination required when computing a value of a given type, and preserves the value i.e. it is an identity function.

```
type Strategy a = a -> Eval a

data Eval a = Done a
```

We provide a simple function to extract a value from `Eval`:

```
runEval :: Eval a -> a
runEval (Done a) = a
```

The `return` operator from the `Eval` monad will introduce a value into the monad:

```
return :: a -> Eval a
return x = Done x
```

Applying Strategies

`using` applies a strategy to a value, e.g.

```
using :: a -> Strategy a -> a
using x s = runEval (s x)
```

Example

A typical GpH function looks like this:

```
somefun x y = someexpr `using` somestrat
```

Applying Strategies

`using` applies a strategy to a value, e.g.

```
using :: a -> Strategy a -> a
using x s = runEval (s x)
```

Example

A typical GpH function looks like this:

```
somefun x y = someexpr `using` somestrat
```

Simple Strategies

Simple strategies can now be defined.

`r0` performs no reduction at all. Used, for example, to evaluate only the first element but not the second of a pair.

`rseq` reduces its argument to Weak Head Normal Form (WHNF).

`rpar` sparks its argument.

```
r0 :: Strategy a
r0 x = Done x

rseq :: Strategy a
rseq x = x `pseq` Done x

rpar :: Strategy a
rpar x = x `par` Done x
```

Controlling Evaluation Order

We control evaluation order by using a monad to sequence the application of strategies.

So our parallel factorial can be written as:

Example (Parallel factorial)

```
pfact' :: Integer -> Integer -> Integer
pfact' m n
  | m == n     = m
  | otherwise = (left * right) `using` strategy
      where mid   = (m + n) `div` 2
            left  = pfact' m mid
            right = pfact' (mid+1) n
            strategy result = do
                                rpar left
                                rseq right
                                return result
```

Controlling Evaluation Degree - The `DeepSeq` Module

Both `r0` and `rseq` control the evaluation degree of an expression.

It is also often useful to reduce an expression to *normal form* (NF), i.e. a form that contains *no* redexes. We do this using the `rnf` strategy in a type class.

As NF and WHNF coincide for many simple types such as `Integer` and `Bool`, the default method for `rnf` is `rwhnf`.

```
class NFData a where
  rnf ::  a -> ()
  rnf x = x `pseq` ()
```

We define `NFData` instances for many types, e.g.

```
instance NFData Int
instance NFData Char
instance NFData Bool
```

Evaluation Degree Strategies

We can define `NFData` for type constructors, e.g.

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs
```

We can define a `deepseq` operator that fully evaluates its first argument:

```
deepseq :: NFData a => a -> b -> b
deepseq a b = rnf a `seq` b
```

Reducing all of an expression with `rdeepseq` is by far the most common evaluation degree strategy:

```
rdeepseq :: NFData a => Strategy a
rdeepseq x = x `deepseq` Done x
```

Combining Strategies

As strategies are simply functions they can be combined using the full power of the language, e.g. passed as parameters or composed.

`dot` composes two strategies on the same type:

```
dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1
```

`evalList` sequentially applies strategy `s` to every element of a list:

Example (Parametric list strategy)

```
evalList :: Strategy a -> Strategy [a]
evalList s []     = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

Data Parallel Strategies

Often coordination follows the data structure, e.g. a thread is created for each element of a data structure.

For example `parList` applies a strategy to every element of a list in parallel using `evalList`

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

`parMap` is a higher order function using a strategy to specify data-oriented parallelism over a list.

```
parMap strat f xs = map f xs `using` parList strat
```

Control-oriented Parallelism

Example (Strategic quicksort)

```
quicksortS []       = []
quicksortS [x]      = [x]
quicksortS (x:xs)   =
  losort ++ (x:hisort) `using` strategy
    where
      losort = quicksortS [y|y <- xs, y < x]
      hisort = quicksortS [y|y <- xs, y >= x]
      strategy res = do
                       (rpar `dot` rdeepseq) losort
                       (rpar `dot` rdeepseq) hisort
                        rdeepseq res
```

Note how the coordination code is cleanly separated from the computation.

Thread Granularity

Using semi-explicit parallelism, programs often have massive, fine-grain parallelism, and several techniques are used to increase thread granularity.

It is only worth creating a thread if the *cost of the computation will outweigh the overheads* of the thread, including

- communicating the computation
- thread creation
- memory allocation
- scheduling

It may be necessary to transform the program to achieve good parallel performance, e.g. to improve thread granularity.
Thresholding: in divide and conquer programs, generate parallelism only up to a certain threshold, and when it is reached, solve the small problem sequentially.

Thread Granularity

Using semi-explicit parallelism, programs often have massive, fine-grain parallelism, and several techniques are used to increase thread granularity.

It is only worth creating a thread if the *cost of the computation will outweigh the overheads* of the thread, including

- communicating the computation
- thread creation
- memory allocation
- scheduling

It may be necessary to transform the program to achieve good parallel performance, e.g. to improve thread granularity.
Thresholding: in divide and conquer programs, generate parallelism only up to a certain threshold, and when it is reached, solve the small problem sequentially.

Threshold Factorial

Example (Strategic factorial with threshold)

```
pfactThresh :: Integer -> Integer -> Integer
pfactThresh n t = pfactThresh' 1 n t

-- thresholding version
pfactThresh' :: Integer -> Integer -> Integer -> Integer
pfactThresh' m n t
  | (n-m) <= t = product [m..n]   -- seq solve
  | otherwise = (left * right) `using` strategy
      where mid   = (m + n) `div` 2
            left  = pfactThresh' m mid t
            right = pfactThresh' (mid+1) n t
            strategy result = do
                                rpar left
                                rseq right
                                return result
```

Chunking Data Parallelism

Evaluating individual elements of a data structure may give too fine thread granularity, whereas evaluating many elements in a single thread give appropriate granularity. The number of elements (the size of the chunk) can be tuned to give good performance.

It's possible to do this by changing the computational part of the program, e.g. replacing

```
parMap rdeepseq fact [12 .. 30]
```

with

```
concat (parMap rdeepseq
          (map fact) (chunk 5 [12 .. 30]))

chunk :: Int -> [a] -> [[a]]
chunk _ [] = [[]]
chunk n xs = y1 : chunk n y2
  where
    (y1, y2) = splitAt n xs
```

Strategic Chunking

Rather than change the computational part of the program, it's better to change only the strategy.

We can do so using the `parListChunk` strategy which applies a strategy `s` sequentially to sublists of length `n`:

```
map fact [12 .. 30] `using` parListChunk 5 rdeepseq
```

Uses Strategy library functions:

```
parListChunk :: Int -> Strategy [a] -> Strategy [a]
parListChunk n s =
  parListSplitAt n s (parListChunk n s)

parListSplitAt :: Int -> Strategy [a]
                  Strategy [a] -> Strategy [a]
parListSplitAt n stratPref stratSuff =
 evalListSplitAt n (rpar `dot` stratPref)
                   (rpar `dot` stratSuff)
```

Strategic Chunking

Rather than change the computational part of the program, it's better to change only the strategy.

We can do so using the `parListChunk` strategy which applies a strategy `s` sequentially to sublists of length `n`:

```
map fact [12 .. 30] `using` parListChunk 5 rdeepseq
```

Uses Strategy library functions:

```
parListChunk :: Int -> Strategy [a] -> Strategy [a]
parListChunk n s =
  parListSplitAt n s (parListChunk n s)

parListSplitAt :: Int -> Strategy [a]
                  Strategy [a] -> Strategy [a]
parListSplitAt n stratPref stratSuff =
 evalListSplitAt n (rpar `dot` stratPref)
                   (rpar `dot` stratSuff)
```

```
evalListSplitAt :: Int -> Strategy [a] ->
                   Strategy [a] -> Strategy [a]
 evalListSplitAt n stratPref stratSuff [] = return []
 evalListSplitAt n stratPref stratSuff xs
  = do
      ys' <- stratPref ys
      zs' <- stratSuff zs
      return (ys' ++ zs')
      where
        (ys,zs) = splitAt n xs
```

Systematic Clustering

Sometimes we require to aggregate collections in a way that cannot be expressed using only strategies. We can do so systematically using the `Cluster` class:

- `cluster n` maps the collection into a collection of collections each of size `n`
- `decluster` retrieves the original collection
  `decluster . cluster == id`
- `lift` applies a function on the original collection to the clustered collection

```
class (Traversable c, Monoid a) => Cluster a c where
  cluster    :: Int -> a -> c a
  decluster  :: c a -> a
  lift       :: (a -> b) -> c a -> c b

  lift = fmap        -- c is a Functor, via Traversable
  decluster = fold   -- c is Foldable,  via Traversable
```

An instance for lists requires us only to define `cluster`

```
instance Cluster [a] [] where
  cluster = chunk
```

A Strategic Div&Conq Skeleton

```
divConq :: (a -> b)                -- compute the result
        -> a                       -- the value
        -> (a -> Bool)             -- threshold reached?
        -> (b -> b -> b)           -- combine results
        -> (a -> Maybe (a,a)) -- divide
        -> b
divConq f arg threshold conquer divide = go arg
 where
  go arg =
    case divide arg of
     Nothing      -> f arg
     Just (l0,r0) -> conquer l1 r1 `using` strat
    where
     l1 = go l0
     r1 = go r0
     strat x = do r l1; r r1; return x
               where r | threshold arg = rseq
                       | otherwise     = rpar
data Maybe a = Nothing | Just a
```

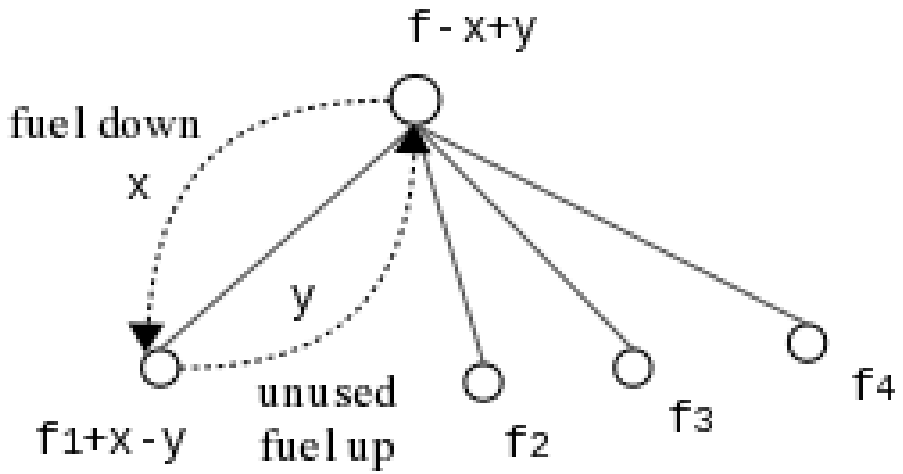
Summary

Evaluation strategies in GpH

- use laziness to *separate computation from coordination*
- use the `Eval` monad to specify evaluation order
- use overloaded functions (`NFData`) to specify the evaluation-degree
- provide high level abstractions, e.g. `parList, parSqMatrix`
- are functions in algorithmic language $\Rightarrow$
  - comprehensible,
  - can be combined, passed as parameters etc,
  - extensible: write application-specific strategies, and
  - can be defined over (almost) any type
- general: pipeline, d&c, data parallel etc.
- Capable of expressing complex coordination, e.g. embedded parallelism, `Clustering`, skeletons

For a list of (parallel) Haskell exercises with usage instructions see:
`http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/tutorial0.html#gph`

Fuel-based parallelism with give-back using circularity



- The resource of “fuel” is used to limit the amount of parallelism when traversing a data structure
- It is passed down from the root of the tree
- It is given back if the tree is empty or fuel is unused
- The give-back mechanism is implemented via *circularity*

Fuel-based parallelism with give-back using circularity

Listing 1: Fuel with giveback annotation

```
1  -- | Fuel with giveback annotation
2  annFuel_giveback::Fuel->QTree tl
3                       ->AnnQTree Fuel tl
4  annFuel_giveback f t = fst $ ann (fuelL f) t
5   where
6    ann::FuelL->QTree tl->(AnnQTree Fuel tl,FuelL)
7    ann f_in E                  = (E,f_in)
8    ann f_in (L x)              = (L x,f_in)
9    ann f_in (N (Q a b c d)) =
10    (N (AQ (A (length f_in)) a' b' c' d'),emptyFuelL)
11    where
12     (f1_in:f2_in:f3_in:f4_in:_) =
13       fuelsplit_unitlist _numSubnodes f_in
14     (a', f1_out ) = ann (f1_in++ f4_out ) a
15     (b', f2_out ) = ann (f2_in++ f1_out ) b
16     (c', f3_out ) = ann (f3_in++ f2_out ) c
17     (d', f4_out ) = ann (f4_in++ f3_out ) d
```

Further Reading & Deeper Hacking

- S. Marlow and P. Maier and H-W. Loidl and M.K. Aswad and P. Trinder, *"Seq no more: Better Strategies for Parallel Haskell"*. In *Haskell'10 — Haskell Symposium*, Baltimore MD, U.S.A., September 2010. ACM Press.
  `http://www.macs.hw.ac.uk/~dsg/projects/gph/papers/abstracts/new-strategies.html`
- Prabhat Totoo, Hans-Wolfgang Loidl. *"Lazy Data-Oriented Evaluation Strategies"*. In *FHPC 2014: The 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing*, Gothenburg, Sweden, September, 2014.
  `http://www.macs.hw.ac.uk/~dsg/projects/gph/papers/abstracts/fhpc14.html`
- *"Parallel and concurrent programming in Haskell"*, by Simon Marlow. O'Reilly, 2013. ISBN: 9781449335946.

Further Reading & Deeper Hacking

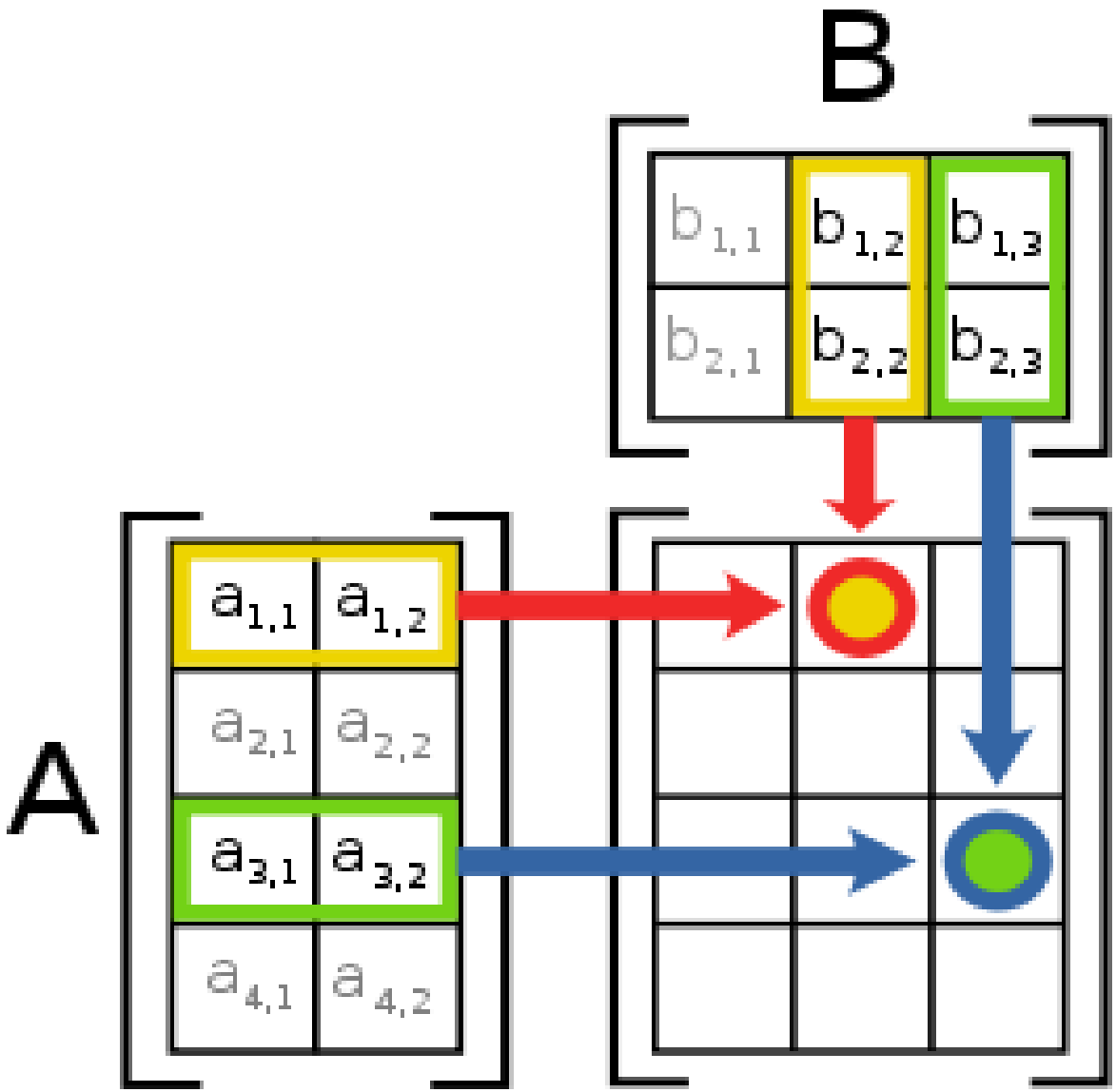
- An excellent site for learning (sequential) Haskell is:
  `https://www.fpcomplete.com/school`
- Glasgow parallel Haskell web page:
  `http://www.macs.hw.ac.uk/~dsg/gph`
- Our course on parallel technologies covers GpH in more detail and has more exercises:
  `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP`
- Specifically, for a list of (parallel) Haskell exercises with usage instructions see:
  `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/tutorial0.html#gph`

Case study: Parallel Matrix Multiplication

As an example of a parallel program lets consider: matrix multiplication.

Problem If matrix A is an $m \times n$ matrix $[a_{ij}]$ and B is an $n \times p$ matrix $[b_{ij}]$, then the product is an $m \times p$ matrix C where $C_{ik} = \Sigma_{j=1}^{n} a_{ij} b_{jk}$

Matrix Multiplication



[0]Picture from `http://en.wikipedia.org/wiki/Matrix_multiplication`

Sequential Implementation

```
-- Type synonyms
type Vec a = [a]
type Mat a = Vec (Vec a)

-- vector multiplication ('dot-product')
mulVec :: Num a => Vec a -> Vec a -> a
u `mulVec` v = sum (zipWith (*) u v)

-- matrix multiplication, in terms of vector multiplic
mulMat :: Num a => Mat a -> Mat a -> Mat a
a `mulMat` b =
  [[u `mulVec` v | v <- bt ] | u <- a]
     where bt = transpose b
```

Parallel Implementation

1st attempt: parallelise every element of the result matrix, or both '`map`s'

```
mulMatPar :: (NFData a, Num a) =>
                  Mat a -> Mat a -> Mat a
mulMatPar a b = (a `mulMat` b) `using` strat
 where
  strat m = parList (parList rdeepseq) m
```

Easy to get a first parallel version.
Unlikely to give good performance straight away.
Some performance tuning is necessary (as with all parallel programming activities).

Parallel Implementation

1st attempt: parallelise every element of the result matrix, or both '`map`s'

```
mulMatPar :: (NFData a, Num a) =>
                  Mat a -> Mat a -> Mat a
mulMatPar a b = (a `mulMat` b) `using` strat
 where
  strat m = parList (parList rdeepseq) m
```

Easy to get a first parallel version.
Unlikely to give good performance straight away.
Some performance tuning is necessary (as with all parallel programming activities).

Shared-Memory Results

600 x 600 matrices on an 8-core shared memory machine (Dell PowerEdge).

Compile with profiling; run on 4 cores; view results

```
% ghc --make -O2 -threaded -eventlog
%   -o MatMultPM MatMultPM.hs
% ./MatMultPM 600 90 20 20 13 +RTS -N7 -sstderr -ls
% threadscope MatMultPM.eventlog
```

No. Cores	Runtime (s)	Relative Speedup	Absolute Speedup
Seq	56.1		1.0
1	62.6	1.0	*0.89*
2	56.9	*1.10*	0.99
4	59.7	1.04	0.95
7	60.2	1.04	0.96

Improving Granularity

Currently parallelise both maps (outer over columns, inner over rows)

Parallelising *only the outer*, and performing the inner sequentially will *increase thread granularity*.

```
mulMatParRow :: (NFData a, Num a) =>
                Mat a -> Mat a -> Mat a
mulMatParRow a b =
  (a `mulMat` b) `using` strat
    where
      strat m = parList rdeepseq m
```

Granularity can be further increased by ‘row clustering’, i.e. evaluating `c` rows in a single thread, e.g.

```
mulMatParRows :: (NFData a, Num a) =>
  Int -> Mat a -> Mat a -> Mat a
mulMatParRows m a b =
  (a `mulMat` b) `using` strat
    where
      strat m = parListChunk c rdeepseq m
```

Clustering (or chunking) is a common technique for increase the performance of data parallel programs.

Granularity can be further increased by 'row clustering', i.e. evaluating `c` rows in a single thread, e.g.

```
mulMatParRows :: (NFData a, Num a) =>
  Int -> Mat a -> Mat a -> Mat a
mulMatParRows m a b =
  (a `mulMat` b) `using` strat
    where
      strat m = parListChunk c rdeepseq m
```

Clustering (or chunking) is a common technique for increase the performance of data parallel programs.

Shared-Memory Row-Clustered Results

600 x 600 matrices with clusters of 90 rows:

No. Cores	Runtime (s)	Relative Speedup	Absolute Speedup
Seq	56.1		1.0
1	60.4	1.0	0.93
2	31.4	1.9	1.8
4	18.0	3.4	3.4
7	9.2	6.6	*6.6*

Algorithmic Improvements

Using blockwise clustering (a.k.a. Gentleman's algorithm) reduces communication as only part of matrix B needs to be communicated.

N.B. Prior to this point we have preserved the computational part of the program and simply added strategies. Now additional computational components are added to cluster the matrix into blocks size `m` times `n`.

```
mulMatParBlocks :: (NFData a, Num a) =>
  Int -> Int -> Mat a -> Mat a -> Mat a
mulMatParBlocks m n a b =
 (a `mulMat` b) `using` strat
  where
   strat x = return (unblock (block m n x
               `using` parList rdeepseq))
```

Algorithmic changes can drastically improve parallel performance, e.g. by reducing communication or by improving data locality.

Algorithmic Improvements

Using blockwise clustering (a.k.a. Gentleman's algorithm) reduces communication as only part of matrix B needs to be communicated.

N.B. Prior to this point we have preserved the computational part of the program and simply added strategies. Now additional computational components are added to cluster the matrix into blocks size `m` times `n`.

```
mulMatParBlocks :: (NFData a, Num a) =>
  Int -> Int -> Mat a -> Mat a -> Mat a
mulMatParBlocks m n a b =
 (a `mulMat` b) `using` strat
  where
   strat x = return (unblock (block m n x
               `using` parList rdeepseq))
```

Algorithmic changes can drastically improve parallel performance, e.g. by reducing communication or by improving data locality.

`block` clusters a matrix into a matrix of matrices, and `unblock` does the reverse.

```
block :: Int -> Int -> Mat a -> Mat (Mat a)
block m n = map f . chunk m where
  f :: Mat a -> Vec (Mat a)
  f = map transpose . chunk n . transpose

-- Left inverse of @block m n@.
unblock :: Mat (Mat a) -> Mat a
unblock = unchunk . map g where
  g :: Vec (Mat a) -> Mat a
  g = transpose . unchunk . map transpose
```

Results from the Tuned Parallel Version

600 x 600 matrices with block clusters: 20 x 20

No. Cores	Runtime (s)	Relative Speedup	Absolute Speedup
Seq	56.1		1.0
1	60.4	1.0	0.93
2	26.9	*2.2*	2.1
4	14.1	*4.2*	3.9
7	8.4	*7.2*	*6.7*

Parallel Threadscope Profiles

For parallelism profiles compile with option `-eventlog`

```
ghc -O2 -rtsopts -threaded -eventlog
    -o parsum_thr_1 parsum.hs
```

then run with runtime-system option `-ls`

```
./parsum_thr_1 90M 100 +RTS -N6 -ls
```

and visualise the generated eventlog profile like this:

```
/home/pt114/.cabal/bin/threadscope parsum_thr_1.eventlog
```

You probably want to do this on small inputs, otherwise the eventlog file becomes huge!

Parallel Threadscope Profiles

For parallelism profiles compile with option `-eventlog`

```
ghc -O2 -rtsopts -threaded -eventlog
    -o parsum_thr_1 parsum.hs
```

then run with runtime-system option `-ls`

```
./parsum_thr_1 90M 100 +RTS -N6 -ls
```

and visualise the generated eventlog profile like this:

```
/home/pt114/.cabal/bin/threadscope parsum_thr_1.eventlog
```

You probably want to do this on small inputs, otherwise the eventlog file becomes huge!

Parallel Threadscope Profiles