Parallel Programming

Engineering a parallel program entails specifying

- **computation**: a correct, efficient algorithm
- **coordination**: arranging the computations to achieve “good” parallel behaviour. Metrics include:
  - Speedup, i.e. reduction in execution time, and defined as execution time on 1 processor / time on $n$ processors: $\text{speedup} = t_1/t_n$
  - Efficiency: a speedup of 14 is good on 16 processors, but poor on 100.

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

Many approaches have been proposed to reduce the programmer’s coordination management burden, e.g. skeletons, parallelising compilers, etc.

GpH aims to simplify parallel programming by requiring the programmer to specify only a few key aspects of parallel programming, and leaving the language implementation to automatically manage the rest.

GpH is the dominant parallel programming model for Haskell.
GpH Coordination Primitives

GpH provides parallel composition to hint that an expression may usefully be evaluated by a parallel thread. Similar to || in FSP.

We say x is sparked, and if there is an idle processor a thread may be created to evaluate it.

\[ x \text{ 'par' } y \Rightarrow y \]

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. Similar to -> in FSP.

\[ x \text{ 'pseq' } y \Rightarrow y \]

Introducing Parallelism: a GpH Factorial

Factorial is a classic divide and conquer algorithm.

\[
pfact n = pfact' 1 n
\]

\[
 pfact' :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}
\]

\[
 pfact' m n
\]

\[
 | m == n \Rightarrow m
\]

| otherwise = left 'par' right 'pseq' (left + right)

where mid = (m + n) 'div' 2

\[
 \text{left} = pfact' m \text{mid}
\]

\[
 \text{right} = pfact' (\text{mid}+1) n
\]

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.

\[
 | \text{otherwise} = \text{left 'par' (left * right)}
\]

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!

\[
 \text{quicksortN} :: (\text{Ord a}) \Rightarrow [a] \rightarrow [a]
\]

\[
 \text{quicksortN} [] = []
\]

\[
 \text{quicksortN} [x] = [x]
\]

\[
 \text{quicksortN} (x:xs) =
\]

\[
 \text{losort 'par'}
\]

\[
 \text{hisort 'par'}
\]

\[
 \text{losort} \text{ ++ (x:hisort)}
\]

where

\[
 \text{losort} = \text{quicksortN} [y | y <- xs, y < x]
\]

\[
 \text{hisort} = \text{quicksortN} [y | y <- xs, y >= x]
\]
Controlling Evaluation Degree II

Forcing the evaluation of the sublists gives the desired behaviour:

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

```haskell
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, and each composition of datatypes, e.g. list of lists.

GpH Coordination Aspects

To specify parallel coordination 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.*
Applying Strategies

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

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

A typical GpH function looks like:

```haskell
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.

```haskell
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:

```haskell
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`.

```haskell
class NFData a where
  rnf :: a -> ()
  rnf x = x 'seq' ()
```

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

```haskell
instance NFData Int
instance NFData Char
instance NFData Bool
```
We can define `NFData` for type constructors, e.g.

```haskell
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:

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

**Evaluation Degree Strategies**

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

```haskell
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:

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

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

```haskell
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`:

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

For tuples, `parTuple2` evaluates both elements in parallel:

```haskell
parTuple2 :: Strategy a -> Strategy b -> Strategy (a,b)
parTuple2 strat1 strat2 = evalTuple2 (rpar ‘dot’ strat1) (rpar ‘dot’ strat2)
```
Data-oriented Parallelism

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

parMap rdeepseq fact [12 .. 30]

Exercise: How many threads are created by the example above?

Control-oriented Parallelism

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]
        strat res = do
            (rpar 'dot' rdeepseq) losort
            (rpar 'dot' rdeepseq) hisort
            rdeepseq res

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

Pipeline Parallelism

In pipelined parallelism stream-processing functions are composed together, each function consumes the stream of values constructed by the previous function and producing new values for the next.

This kind of parallelism is easily expressed in a non-strict language by function composition. The non-strict semantics automatically manages synchronisation between pipeline stages.

(.||) :: (b -> c) -> Strategy b -> (a -> b) -> (a ->
    (b -> c) s g = \ x -> let z = g x 'using' s in
    z 'par' f z

    (parse .|| rdeepseq lex) sometext

If the streams have the same type, we can build a higher order function to capture the pattern:

pipeline :: Strategy a -> a -> [a->a] -> a
pipeline s inp []   = inp
pipeline s inp (f:fs) =
    pipeline s out fs 'using' strat
    where
        out = f inp
        strat res = do
            (rpar 'dot' s) out
        return res

list = pipeline rdeepseq [1..10]
    [map fib, map fact, map (* 2)]
Thread Granularity
Some programs have coarse grain parallelism, i.e. there are only a few threads. The challenge then is to create enough threads to utilise all Processing Elements (PEs).

More commonly programs 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

Improving Fine Granularity
It may be necessary to transform the program to achieve good parallel performance, e.g. to improve thread granularity.

Thresholds
Small tasks can be avoided in divide and conquer programs by not dividing the problem once a threshold is reached, and instead solving the small problem sequentially.

Threshold Factorial
\[
\text{pfactThresh} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{pfactThresh} \ n \ t = \text{pfactThresh'} \ 1 \ n \ t
\]

\[
\text{pfactThresh'} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{pfactThresh'} \ m \ n \ t \\
\begin{cases} 
(n-m) \leq t & \text{product [m..n]} \quad \text{-- seq solve} \\
\text{otherwise} & (\text{left } \ast \text{ right}) \ 'using' \ \text{strategy} \\
\end{cases}
\]

where
\[
\text{mid} = (m + n) \ 'div' \ 2
\]

\[
\text{left} = \text{pfactThresh'} \ m \ \text{mid} \ t
\]

\[
\text{right} = \text{pfactThresh'} \ (\text{mid}+1) \ n \ t
\]

\[
\text{strategy result} = \text{do}
\]

\[
\begin{aligned}
\text{rpar left} \\
\text{rseq right} \\
\text{return result}
\end{aligned}
\]

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

\[
\text{parMap rdeepseq fact [12 .. 30]}
\]

with

\[
\text{concat (parMap rdeepseq (map fact) (splitAtN 5 [12 .. 30]))}
\]

\[
\text{splitAtN :: Int -> [a] -> [[a]]}
\]

\[
\text{splitAtN n [] = []}
\]

\[
\text{splitAtN n xs} = \text{ys} : \text{splitAtN n zs}
\]

where
\[
\text{(ys,zs) = splitAt n xs}
\]
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 \( n \):

```haskell
map fact [12 .. 30] \textquotesingle{}using\textquotesingle{} parListChunk 5 rdeepseq
```

Uses Strategy library functions:

```haskell
parListChunk :: Int \rightarrow\ Strategy \[a\] \rightarrow\ Strategy \[a\]
parListChunk n s =
    parListSplitAt n s (parListChunk n s)

parListSplitAt :: Int \rightarrow\ Strategy \[a\] \rightarrow\ Strategy \[a\]
parListSplitAt n stratPref stratSuff =
    evalListSplitAt n (rpar \textsc{dot} stratPref)
                                (rpar \textsc{dot} stratSuff)
```

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

```haskell
class (Traversable c, Monoid a) \Rightarrow\ Cluster a c where
    cluster :: Int \rightarrow\ a \rightarrow\ c a
    decluster :: c a \rightarrow\ a
    lift :: (a \rightarrow\ b) \rightarrow\ c a \rightarrow\ 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`

```haskell
instance Cluster [a] [] where
    cluster = chunk
```
A Strategic Div&Conq Skeleton

We can capture common patterns of parallel computation as higher order functions, e.g.

\[
\text{divConq} :: (a \to b) \to a \to (a \to \text{Bool}) \to (b \to b \to b) \to (a \to \text{Maybe} (a,a)) \to b
\]

\[
divConq f \text{ arg} \text{ threshold} \text{ conquer} \text{ divide} = \text{go arg}
\]
where

\[
\text{go arg} =
\begin{cases}
\text{Nothing} & \to f \text{ arg} \\
\text{Just} (10, r0) & \to \text{conquer} l1 r1 \text{ 'using' strat}
\end{cases}
\]

\[
\text{strat} x = \text{do} \ r \ l1; \ r \ r1; \ \text{return} \ x
\]
where

\[
\begin{align*}
\text{when } r & | \text{ threshold arg} = \text{rseq} \\
& | \text{ otherwise} = \text{rpar}
\end{align*}
\]

\[
data \text{ Maybe a} = \text{Nothing} \mid \text{Just a}
\]

Embedded Strategies

It is sometimes useful to embed parallelism in a lazy data structure so that parallelism is created on-demand by the consumer.

A \text{parBuffer} n s x yields a list in which evaluation of the \(i\)th element induces parallel evaluation of the \((i+n)\)th element with the first \(n\) elements being evaluated in parallel immediately.

\[
\text{parBuffer} :: \text{Int} \to \text{Strategy } a \to \text{Strategy } [a]
\]

\[
\text{evalBuffer} :: \text{Int} \to \text{Strategy } a \to \text{Strategy } [a]
\]

Evaluation Strategy Summary

- use laziness to separate algorithm from coordination
- use the \text{Eval} monad to specify evaluation order
- use overloaded functions (\text{NFData}) to specify the evaluation-degree
- provide high level abstractions, e.g. \text{parList, parSqMatrix}
- are functions in algorithmic language ⇒
  - 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

Uses of GpH

Many Haskell Programs About 1 in 3 existing functional programs will give acceptable speedups on multicore [TFP10].

Parallel Prototyping A high-level coordination notation like evaluation strategies mean that the programmer can explore alternative parallelisations with relatively little effort. With low-level notations a single parallelisation must be must be designed into the program from the start.

Parallel symbolic applications, e.g. natural language processors, symbolic algebra systems, etc.

Teaching parallel programming