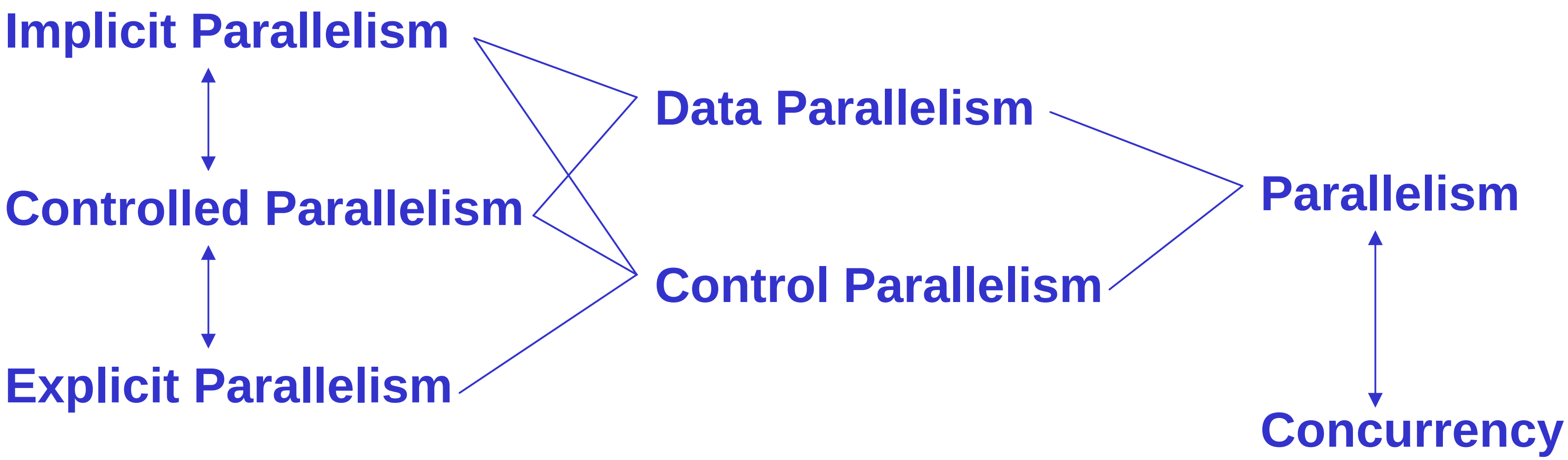
Alternative Concepts:
Parallel Functional Programming

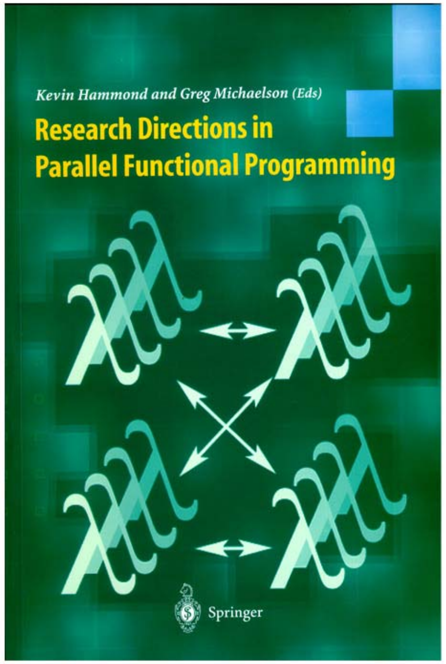


Overview

- Introduction
- From Implicit to Controlled Parallelism
  - Strictness analysis uncovers inherent parallelism
  - Annotations mark potential parallelism
  - Evaluation strategies control dynamic behaviour
- Process-control and Coordination Languages
  - Lazy streams model communication
  - Process nets describe parallel systems
- Data Parallelism
  - Data parallel combinators
  - Nested parallelism

The Book:

Kevin Hammond and Greg Michaelson (Editors):
Research Directions in Parallel Functional Programming
Springer 1999
20 chapters by 27 authors
>= 600 references



Excerpts from the Foreword by S. Peyton Jones

Programming is hard. ... But parallel programming is much, much harder.

...

Functional programming is a radical, elegant, high-level attack on the programming problem. ...

Parallel functional programming is the same, only more so. The rewards are even greater. ...

Parallelism without tears, perhaps? Definitely not. ... Two things have become clear over the last 15 years or so.

First, it is a very substantial task to engineer a parallel functional language implementation....

Second, ... Quite a bit of work needs to go into designing and expressing a parallel algorithm for the application. ... All the interesting work these days is about ... exercising carefully-chosen control over parallel functional programs. ...

Is parallel functional programming any good? If I am honest, I have to say that the jury is still out. ....

Why Parallel Functional Programming Matters

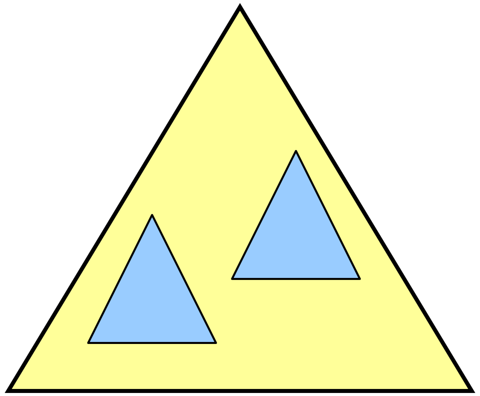
- Hughes 1989: Why Functional Programming Matters
  - ease of program construction
  - ease of function/module reuse
  - simplicity
  - generality through higher-order functions (“functional glue”)

- additional points suggested by experience
  - ease of reasoning / proof
  - ease of program transformation
  - scope for optimisation

- Hammond 1999: additional reasons for the parallel programmer:
  - ease of partitioning a parallel program
  - simple communication model
  - absence from deadlock
  - straightforward semantic debugging
  - easy exploitation of pipelining and other parallel control constructs

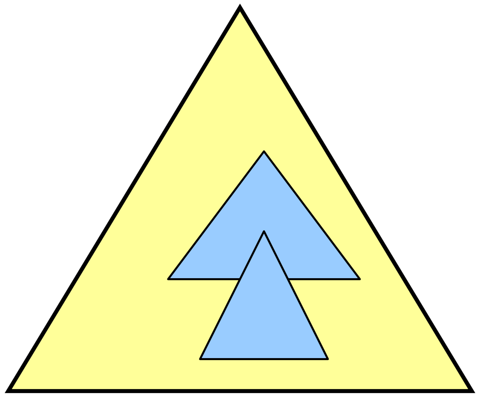
Inherent Parallelism in Functional Programs

- Church Rosser property (confluence) of reduction semantics
  => independent subexpressions can be evaluated in parallel



let f x = e1
g x = e2
in (f 10) + (g 20)

- Data dependencies introduce the need for communication:



let f x = e1
g x = e2
in g (f 10)

----> pipeline parallelism

Further Semantic Properties

- Determinacy: Purely functional programs have the same semantic value when evaluated in parallel as when evaluated sequentially. The value is independent of the evaluation order that is chosen.
  - no race conditions
  - system issues as variations in communication latencies, the intricacies of scheduling of parallel tasks do not affect the result of a program

  Testing and debugging can be done on a sequential machine.

  Nevertheless, performance monitoring tools are necessary on the parallel machine.

- Absence of Deadlock: Any program that delivers a value when run sequentially will deliver the same value then run in parallel.

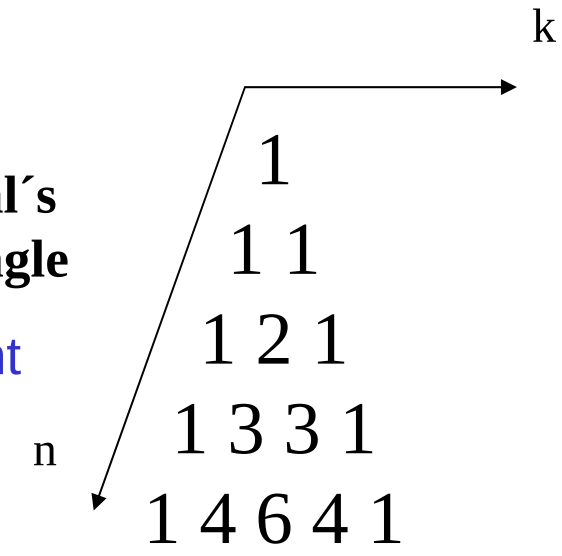
  However, an erroneous program (i.e. one whose result is undefined) may fail to terminate, when executed either sequentially or in parallel.

A Classification

Parallelism	control	data
implicit	automatic parallelisation **annotation-based languages**	data parallel languages
controlled	para-functional programming **evaluation strategies** skeletons	high-level data parallelism
explicit	**process control languages** message passing languages concurrent languages	

Running Examples

Pascal´s Triangle



- binomial coefficients:

```
binom                                :: Int -> Int -> Int
binom n k    | k == 0 && n >= 0   = 1
             | n <   k && n >= 0  = 0
             | n >= k && k >= 0   = binom (n-1) k + binom (n-1) (k-1)
             | otherwise           = error "negative params"
```

- multiplication of sparse matrices with dense vectors:

```
type   SparseMatrix a  = [[(Int,a)]]  -- rows with (col,nz-val) pairs
type   Vector a            = [a]

matvec        :: Num a => SparseMatrix a -> Vector a -> Vector a
matvec m v   = map (sum.map (\ (i,x) -> x * v!!i)) m
```

From Implicit to Controlled Parallelism

Implicit Parallelism (only control parallelism):

- Automatic Parallelisation, Strictness Analysis
- Indicating Parallelism: parallel let, annotations, parallel combinators

semantically transparent parallelism
introduced through low-level language constructs

Controlled Parallelism

- Para-functional programming
- Evaluation strategies

still semantically transparent parallelism
programmer is aware of parallelism
higher-level language constructs

Parallel Combinators

- special projection functions which provide control over the evaluation of their arguments

- e.g. in Glasgow parallel Haskell (GpH):

  par, seq :: a -> b -> b

  where
  - par e1 e2 creates a spark for e1 and returns e2. A spark is a marker that an expression can be evaluated in parallel.
  - seq e1 e2 evaluates e1 to WHNF and returns e2 (sequential composition).

- advantages:
  - simple, annotations as functions (in the spirit of functional programming)
- disadvantages:
  - explicit control of evaluation order by use of seq necessary
  - programs must be restructured

Examples with Parallel Combinators

- binomial coefficients:

```
binom                        :: Int -> Int -> Int
binom n k  | k == 0 && n >= 0 = 1
           | n <   k && n >= 0 = 0
           | n >= k && k >= 0 = let   b1 = binom (n-1) k
                                      b2 = binom (n-1) (k-1)
                                in b2  ‘par‘ b1 ‘seq‘  (b1 + b2)
           | otherwise         = error “negative params”
```

- parallel map:

```
parmap     :: (a-> b) -> [a] -> [b]
parmap  f  [ ]       = [ ]
parmap  f  (x:xs)    = let    fx = (f x)
                              fxs = parmap f xs
                       in fx ‘par‘ fxs ‘seq‘ (fx : fxs)
```

explicit control
of evaluation order

Controlled Parallelism

- parallelism under the control of the programmer
- more powerful constructs
- semi-explicit
  - explicit in the form of special constructs or operations
  - details are hidden within the implementation of these constructs/operations
- no explicit notion of a parallel process
- denotational semantics remains unchanged, parallelism is only a matter of the implementation

- e.g. para-functional programming [Hudak 1986]
  evaluation strategies [Trinder, Hammond, Loidl, Peyton Jones 1998]

Evaluation Strategies

- high-level control of dynamic behavior, i.e. the evaluation degree of an expression and parallelism
- defined on top of parallel combinators par and seq

- An evaluation strategy is a function taking as an argument the value to be computed. It is executed purely for effect. Its result is simply ():

```
type Strategy a      =  a  ->  ()
```

result type → a

"unit" type → ()

The using function allows strategies to be attached to functions:

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

- clear separation of
  - the algorithm specified by a functional program and
  - the specification of its dynamic behavior

Example for Evaluation Strategies

binomial coefficients:

```
binom                              :: Int -> Int -> Int
binom n k  | k == 0 && n >= 0 = 1
           | n <   k && n >= 0 = 0
           | n >= k && k >= 0 = (b1 + b2)  ‘using‘ strat
           | otherwise              = error “negative params”
           where
                b1 = binom (n-1) k
                b2 = binom (n-1) (k-1)
                strat _  =  b2  ‘par‘   b1 ‘seq‘  ()
```

functional program

dynamic behaviour

Evaluation Degrees

- Strategies which specify the degree of evaluation
  - no reduction: r0 :: Strategy a with r0 _ = ( )
  - reduce to weak head normal form:

    rwhnf :: Strategy a with rwhnf x = x `seq` ()
  - reduce to full normal form:

```
class NFData a where
      rnf :: Strategy a
      rnf    = rwhnf      -- default definition
```

- Instance Declarations provide special definitions for data structures:

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

instance (NFData a, NFData b) => (a,b) where
      rnf (a,b)     = rnf a `seq` rnf b `seq` ()
```

Composing Strategies

Strategies are normal higher-order functions, hence

- can be passed as parameters
- composed with other strategies (using function composition etc.)
- etc.

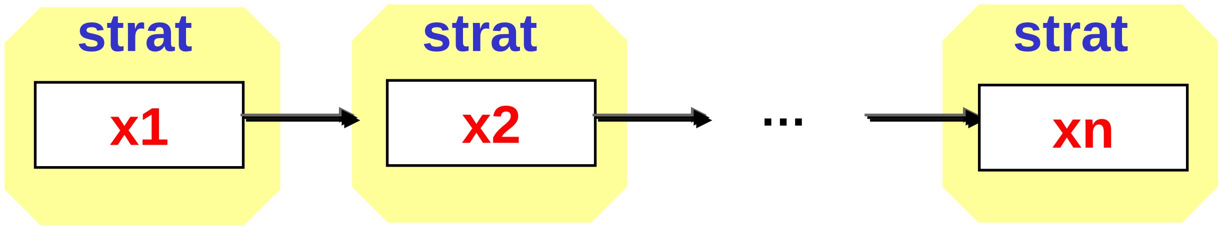
Example:

seqList is a strategy on lists that is parameterised by a strategy on list elements

```
seqList              :: Strategy a -> Strategy [a]
seqList strat [ ]    = ( )
seqList strat (x:xs) = strat x `seq` (seqList strat xs)
```

e.g. seqList r0 — evaluate spine of list
seqList rwhnf — evaluate every element to WHNF

Data-Oriented Parallelism / Parallel Map



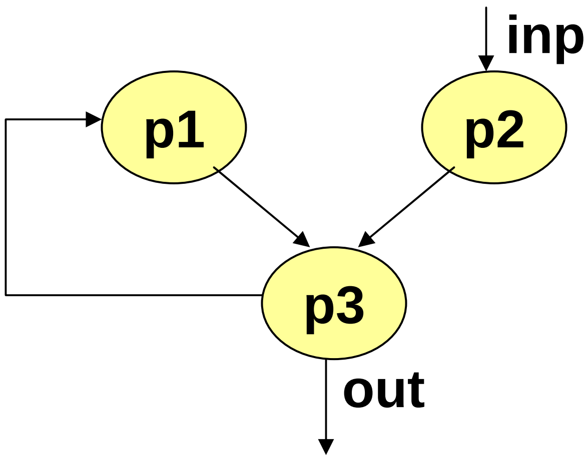

```
parList               :: Strategy a -> Strategy [a]
parList strat []      = ()
parList strat (x:xs)  = strat x `par` (parList strat xs)
```

e.g. parList rwhnf   evaluate each x_i in parallel

```
parMap              ::  Strategy b -> (a -> b) -> [a] -> [b]
parMap  strat  f  xs  =  map f xs `using` parList strat
```

Process-control and Coordination Languages

- Higher-order functions and laziness are powerful abstraction mechanisms which can also be exploited for parallelism:
  - lazy lists can be used to model communication streams
  - higher-order functions can be used to define general process structures or skeletons
- Dynamically evolving process networks can simply be described in a functional framework [Kahn, MacQueen 1977]




```
let  outp2           = p2  inp
     (outp3, out)    = p3 outp1 outp2
     outp1           = p1 outp3
in out
```

Philipps-Universität Marburg

Jost Berthold, Rita Loogen, Steffen Priebe et al.

Acción Integrada 1996-1998

ARC 1999-2001

The Eden Project

Universidad Complutense de Madrid

Yolanda Ortega Mallén

Ricardo Peña Marí

et al.

Heriot-Watt Univ. Edinburgh

Phil Trinder et al.

University of St. Andrews

Kevin Hammond

et al.

Acción Integrada 2000-2002

Parallel Programming at a High Level of Abstraction

parallelism control

- explicit processes
- implicit communication
  (no send/receive)
  - runtime system control
  - stream-based typed communication channels
- disjoint address spaces, distributed memory
- nondeterminism, reactive systems

+

functional language

» polymorphic type system

» pattern matching

» higher order functions

» lazy evaluation

» ...

Eden

parallel functional language

- **computation language: Haskell**
- **coordination language:**
  - + **<u>process abstraction</u>**

    pabs :: Process $(\tau_1,\ldots,\tau_n)$ $(\sigma_1,\ldots,\sigma_m)$

    pabs = process (\ $(i_1,\ldots,i_n)$ -> $(o_1,\ldots,o_m)$

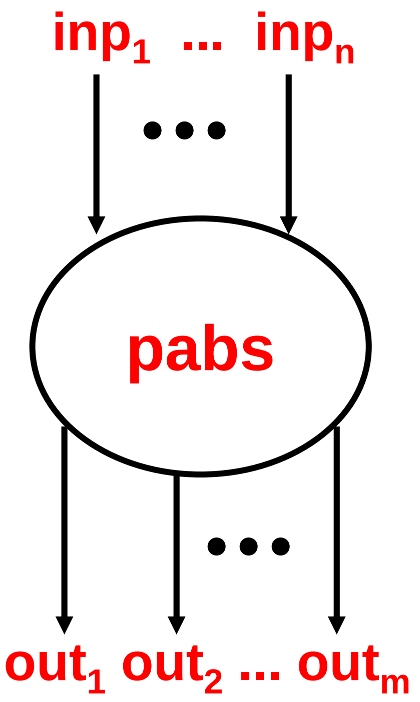
    where eqn_1 ... eqn_k)

  - + **<u>process instantiation</u>**

    (#) :: (Trans a, Trans b) =>
    Process a b -> a -> b

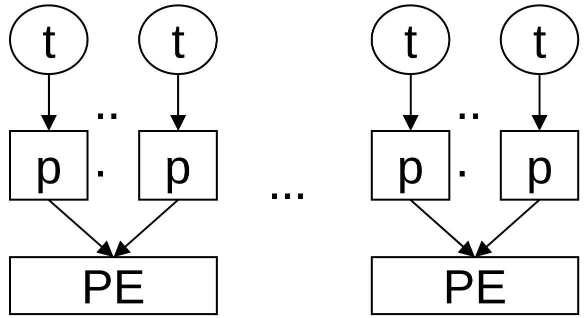
    pabs # $(inp_1,\ldots,inp_n)$ :: $(\sigma_1,\ldots,\sigma_m)$

  - + **...**

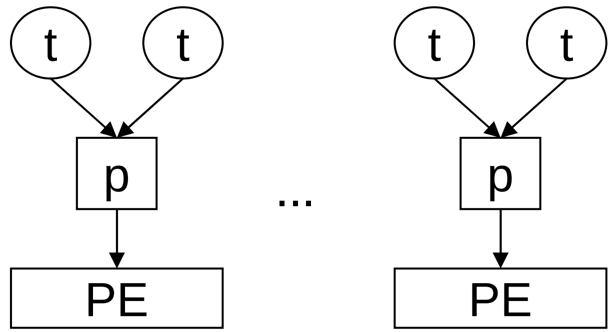


Simple Eden Skeletons

```
parMap :: (Trans t, Trans r) =>
                Process t r -> [t] -> [r]
parMap p ts   = [ p  #  t | t <- ts] `using` spine
```



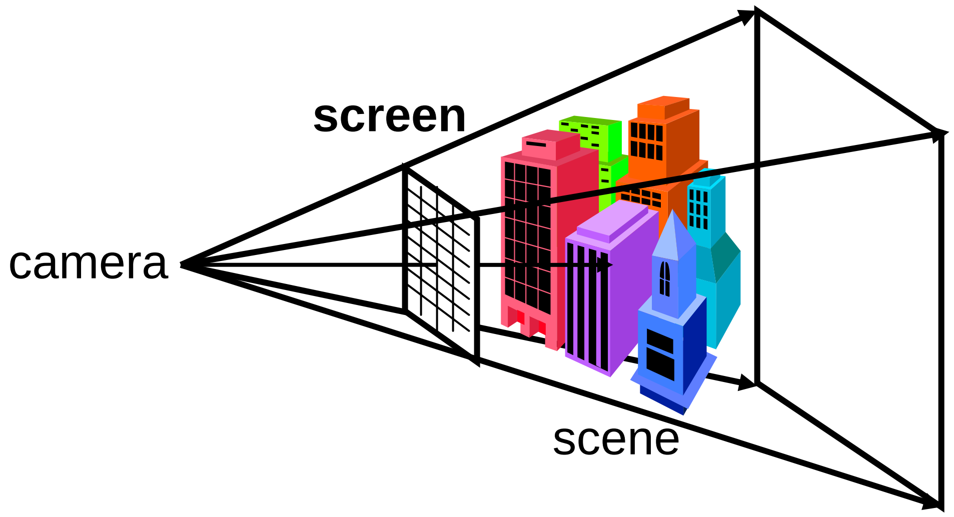

```
farm ::      (Trans t, Trans r) =>
             Int -> (Int -> [t] -> [[t]]) -> ([[r]] -> [r])
             -> Process [t] [r] -> [t] -> [r]
farm np distr combine p ts
             = combine (parMap p (distr np ts))
```




```
map_farm ::  (Trans a, Trans b) =>  (a -> b) -> [a] -> [b]
map_farm  f   = farm noPE shuffle unshuffle  (process f)
```

Eden Example Program

Ray tracer: calculate 2D image of 3D scene



rayTrace :: ScreenSize -> CamPos -> [Object] -> [Impact]

rayTrace scr cameraPos scene

= map_farm (firstImpact scene) allRays

where allRays = generateRays scr cameraPos

Conclusions and Future Work

- language design: various levels of parallelism control and process models
- existing parallel/distributed implementations:
  Clean, GpH, Eden, SkelML, P3L ....
- applications/benchmarks:
  sorting, combinatorial search, n-body, computer algebra, scientific computing ......
- semantics, analysis and transformation:
  strictness, granularity, types and effects, cost analysis ....
- programming methodology:
  skeletons ......