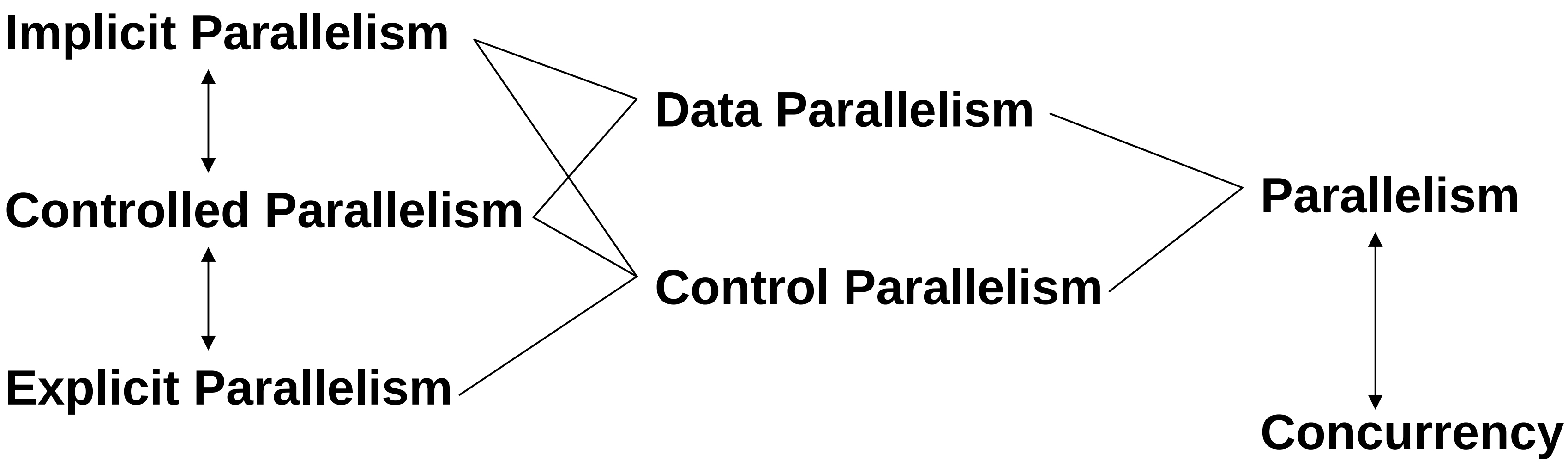
9. *Alternative Konzepte: Parallele funktionale Programmierung*



Kernel Ideas

- **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**

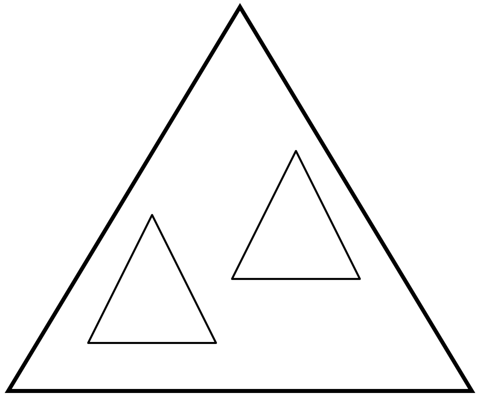
- **Data Parallelism**
  - **Data parallel combinators**
  - **Nested parallelism**

Why Parallel Functional Progr. 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**

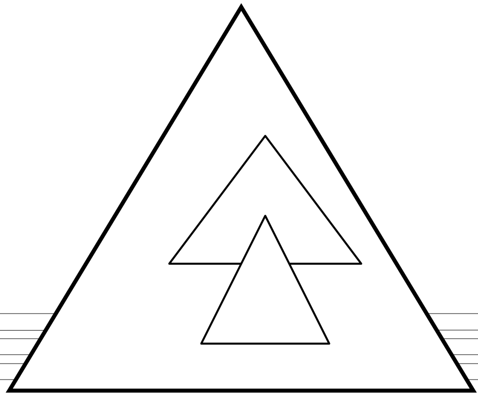
Inherent Parallelism in Functional Programs

- **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	

Examples

- **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**

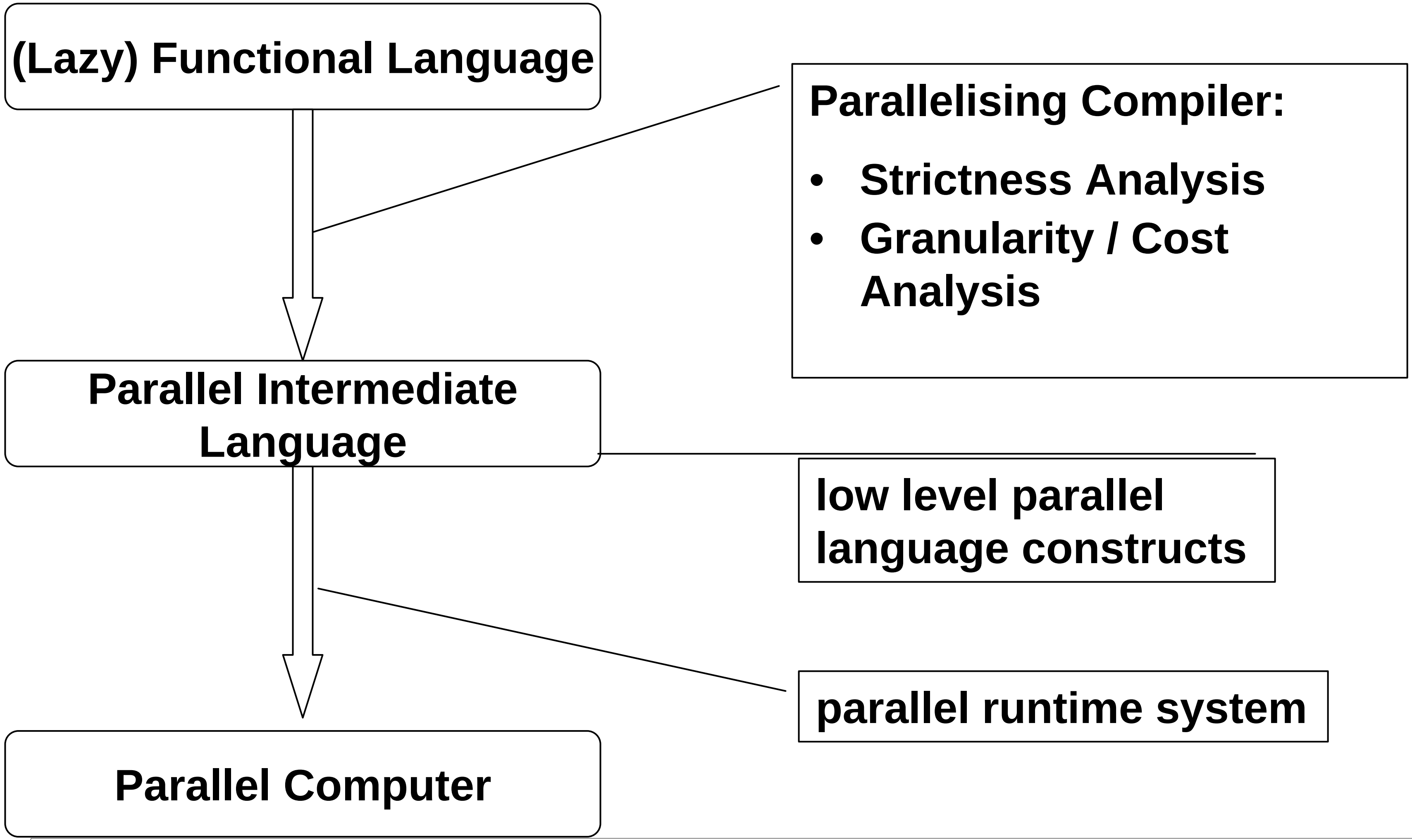
semantically transparent parallelism
introduced through low-level language constructs

Controlled Parallelism

- **Para-functional programming**
- **Evaluation strategies**

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

Automatic Parallelisation



Indicating Parallelism

- **parallel let**
- **annotations**
- **predefined combinators**

}

- semantically transparent
- only advice for the compiler
- do not enforce parallel evaluation

As it is very difficult to detect parallelism automatically, it is common for programmers to indicate parallelism manually.

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 funct. progr.)
- 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‘  ()
```

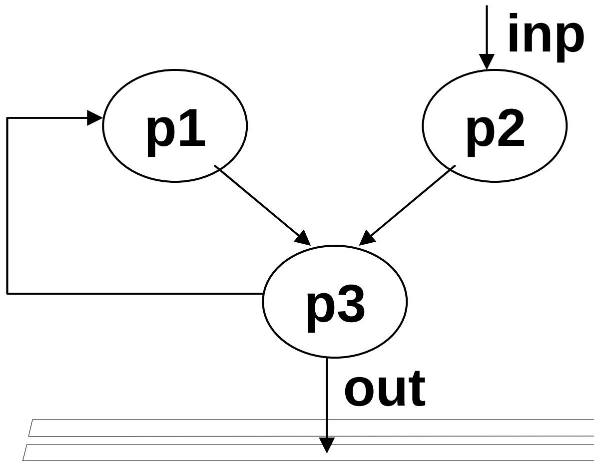
functional program

dynamic behaviour

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
```

Eden:
Parallel Programming at a High Level of Abstraction

parallelism control + **functional language**

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**
» **...**

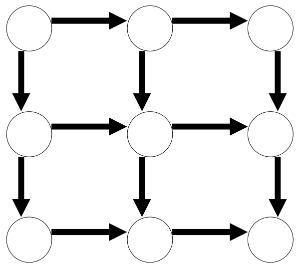
Eden

parallel programming at a high level of abstraction

= Haskell + Coordination

- process definition

```
process        ::    (Trans a, Trans b) =>  (a -> b) -> Process a b
gridProcess    =     process (\ (fromLeft,fromTop) ->
                                   let    ...    in   (toRight, toBottom))
```

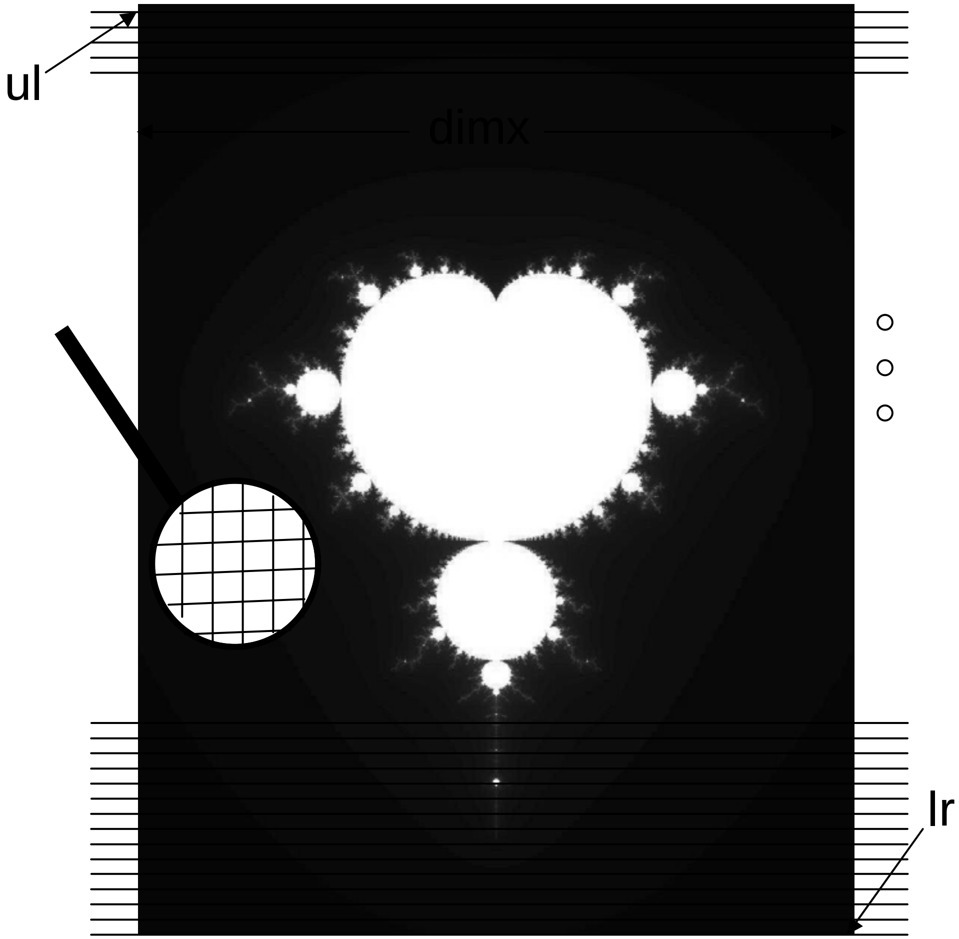


process outputs computed by concurrent threads, lists sent as streams

- process instantiation

```
( # )                  ::     (Trans a, Trans b) =>  Process a b -> a -> b
(outEast, outSouth)    =    gridProcess   #   (inWest,inNorth)
```

Example: Functional Program for Mandelbrot Sets

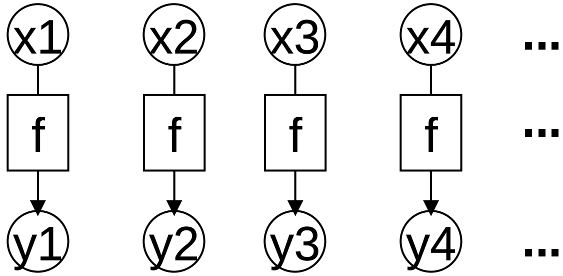


Idea: parallel computation of lines

```
image :: Double -> Complex Double -> Complex Double -> Integer -> String
image threshold ul lr dimx
 = header ++ (  concat $ map xy2col lines  )
 where
   xy2col ::[Complex Double] ->  String
   xy2col line   = concatMap (rgb.(iter threshold (0.0 :+ 0.0) 0)) line
  (dimy, lines)  = coord ul lr dimx
```

Simple Parallelisations of map

```
map :: (a->b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```



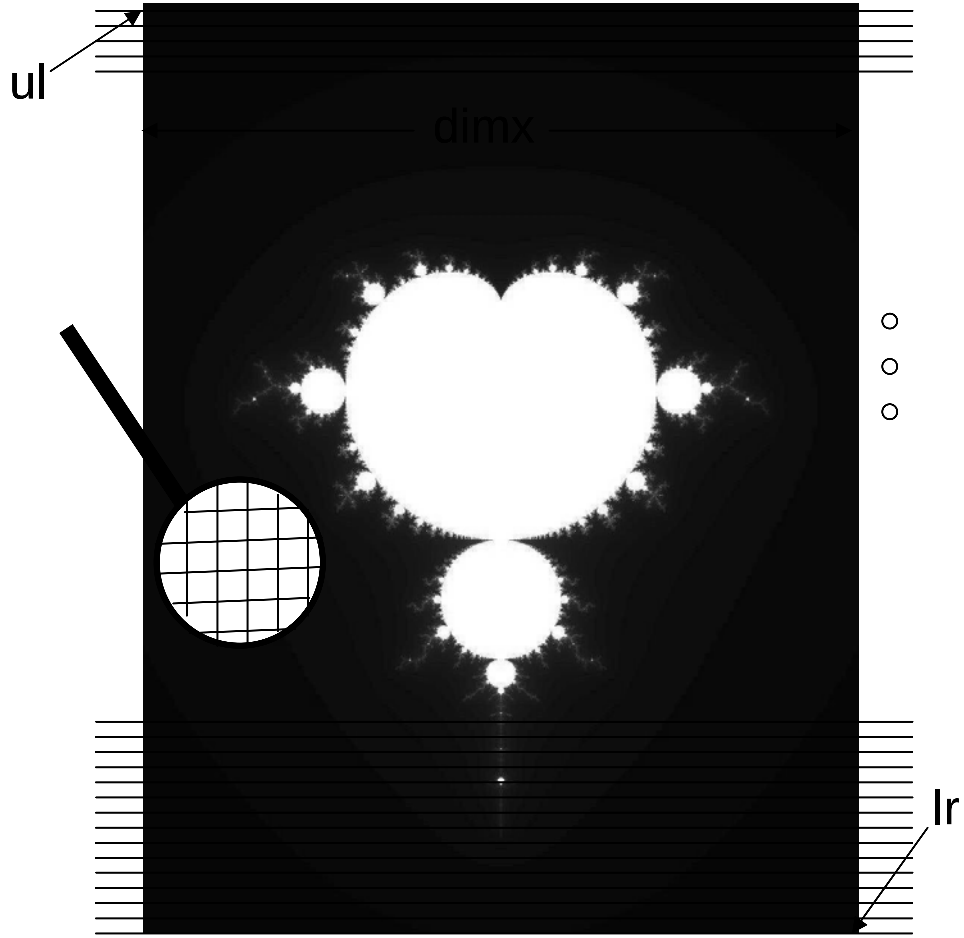

```
parMap ::   (Trans a, Trans b) =>
            (a->b) -> [a] -> [b]
parMap f xs = [ (process f) # x | x <- xs]
                  `using` spine

farm, farmB :: (Trans a, Trans b) =>
               (a->b) -> [a] -> [b]
farm f xs      =  shuffle (parMap (map f)
                        (unshuffle noPe xs))
farmB f xs     =  concat (parMap (map f)
                        (block noPe xs))
```

1 process per list element

1 process per processor with static task distribution

Example: Parallel Functional Program for Mandelbrot Sets



Idea: parallel computation of lines

```
image :: Double -> Complex Double -> Complex Double -> Integer -> String
image threshold ul lr dimx
 = header ++ (  concat $ map xy2col lines  )
 where
   xy2col ::[Complex Double] ->  String
   xy2col line   = concatMap (rgb.(iter threshold (0.0 :+ 0.0) 0)) line
  (dimy, lines)  = coord ul lr dimx
```

Replace map by farm or farmB

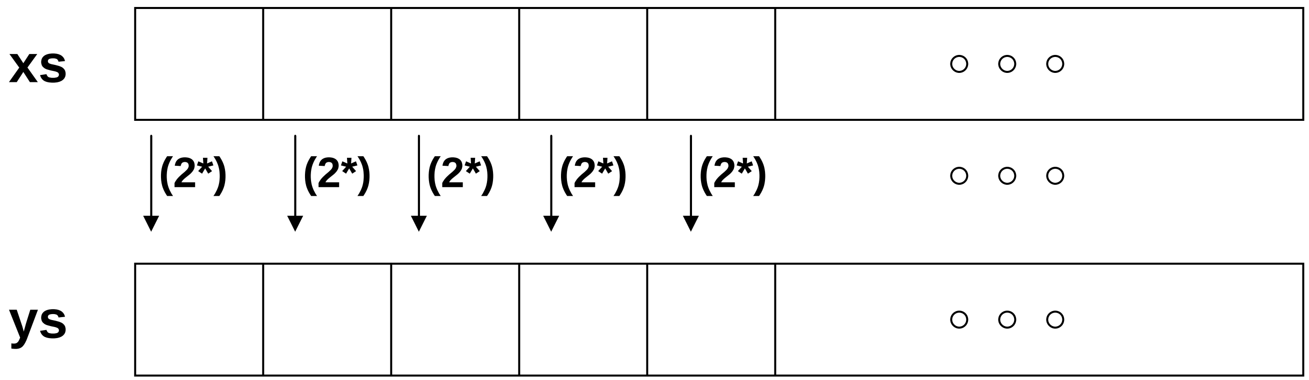
Data Parallelism

[John O'Donnell, Chapter 7 of [Hammond, Michaelson 99]]

Global operations on large data structures are done in parallel by performing the individual operations on singleton elements simultaneously.

The parallelism is determined by the organisation of data structures rather than the organisation of processes.

Example: ys = map (2 ∗) xs



→ **explicit control of parallelism with inherently parallel operations**

→ **naturally scaling with the problem size**

Data-parallel Languages

- main application area: scientific computing
- requirements: efficient matrix and vector operations
  - distributed arrays
  - parallel transformation and reduction operations
- languages
  - imperative:
    - FORTRAN 90: aggregate array operations
    - HPF (High Performance FORTRAN): distribution directives, loop parallelism
  - functional:
    - SISAL (Streams and Iterations in a Single Assignment Language): applicative-order evaluation, forall-expressions, stream-/pipeline parallelism, function parallelism
    - Id, pH (parallel Haskell): concurrent evaluation, I- and M-structures (write-once and updatable storage locations), expression, loop and function parallelism
    - SAC (Single Assignment C): With-loops (dimension-invariant form of array comprehensions)

Finite Sequences

- **simplest parallel data structure**
- **vector, array, list distributed across processors of a distributed-memory multiprocessor**

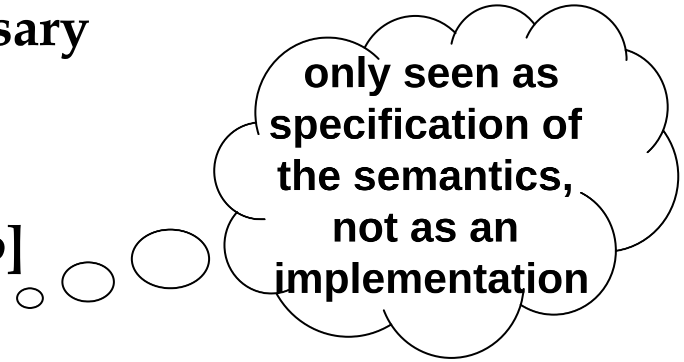
  A finite sequence xs of length k is written as $[x_0, x_1, \ldots x_{k-1}]$.

  For simplicity, we assume that k = N, where N is the number of processor elements. The element x_i is placed in the memory of processor P_i.

- **Lists can be used to represent finite sequences. It is important to remember that such lists**
  - **must have finite length,**
  - **do not allow sharing of sublists, and**
  - **will be computed strictly.**

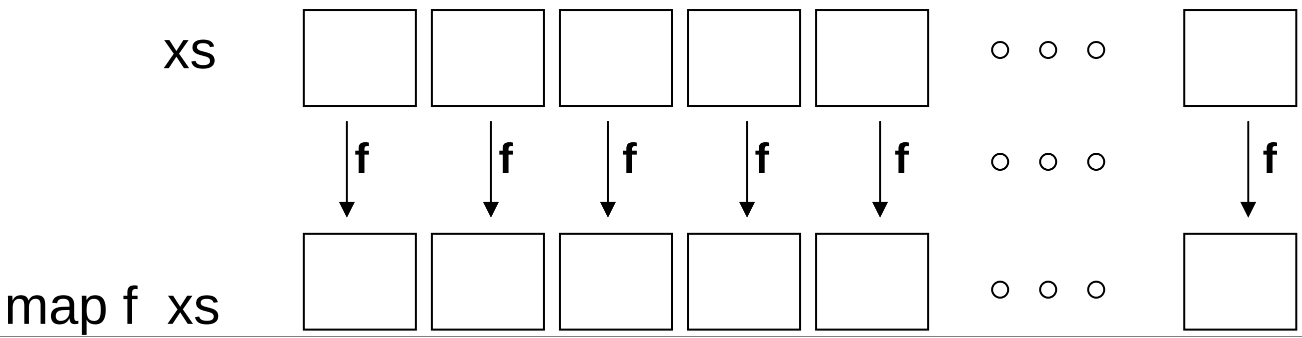
Data Parallel Combinators

- **Higher-order functions are good at expressing data parallel operations:**
  - **flexible and general, may be user-defined**
  - **normal reasoning tools applicable, but special data parallel implementations as primitives necessary**



- **Sequence transformation:**

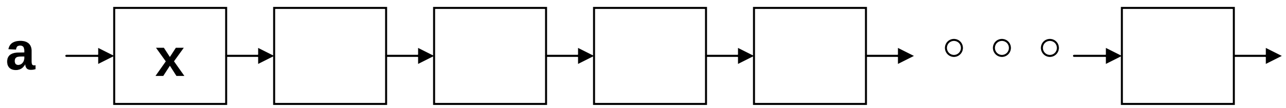
```
map              ::   (a -> b) -> [a] -> [b]
map  f  [ ]      =    [ ]
map  f  (x:xs)   =    (f x) : map  f  xs
```



Communication Combinators

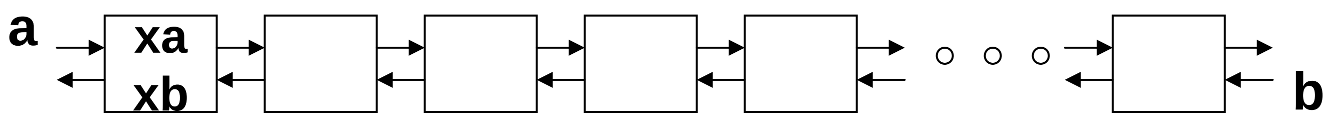
Nearest Neighbour Network

- **unidirectional communication:**




```
shiftr              ::  a -> [a] -> ([a], a)
shiftr  a  [ ]      =   ([ ], a)
shiftr  a  (x:xs)   =   (a:xs',x')
                        where (xs',x') = shiftr x xs
```

- **bidirectional communication:**



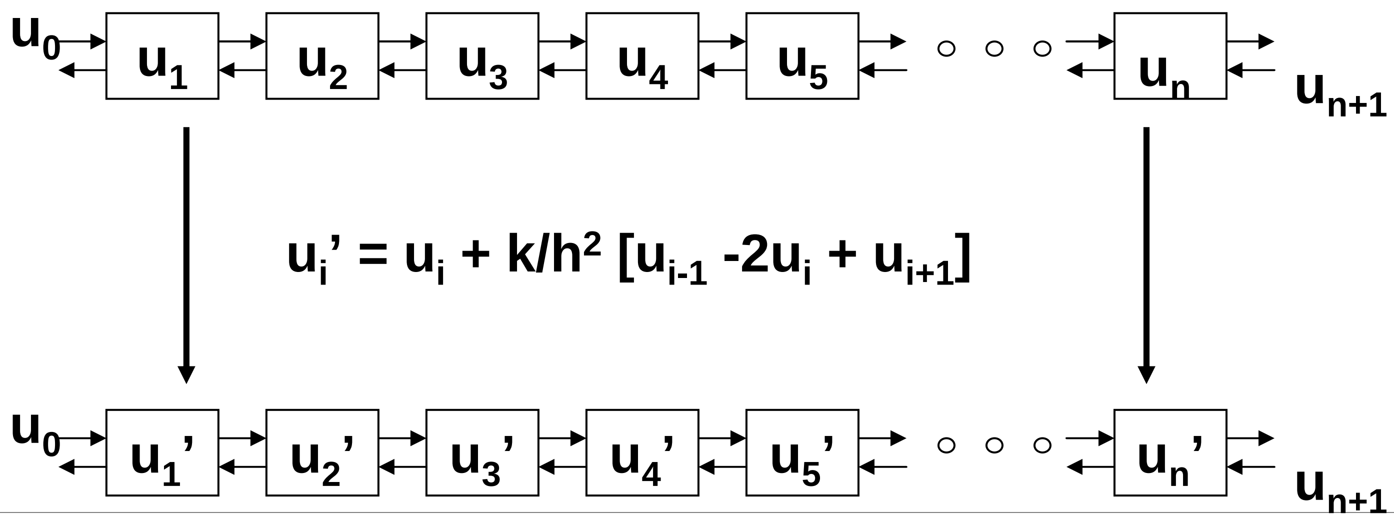

```
shift                       ::  a -> b -> [(a,b)] -> (a,b,[(a,b)])
shift  a  b  [ ]            =   (a,b,[ ])
shift  a  b  ((xa,xb):xs)   =   (a', xb, (a,b'):xs')
                                where   (a', b', xs') = shift xa b xs
```

Example: The Heat Equation

Numerical Solution of the one-dimensional heat equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \text{, for } x \in (0,1) \text{ and } t > 0$$

The continuous interval is represented as a linear sequence of n discrete gridpoints u_i, for $1 \le i \le n$, and the solution proceeds in discrete timesteps:

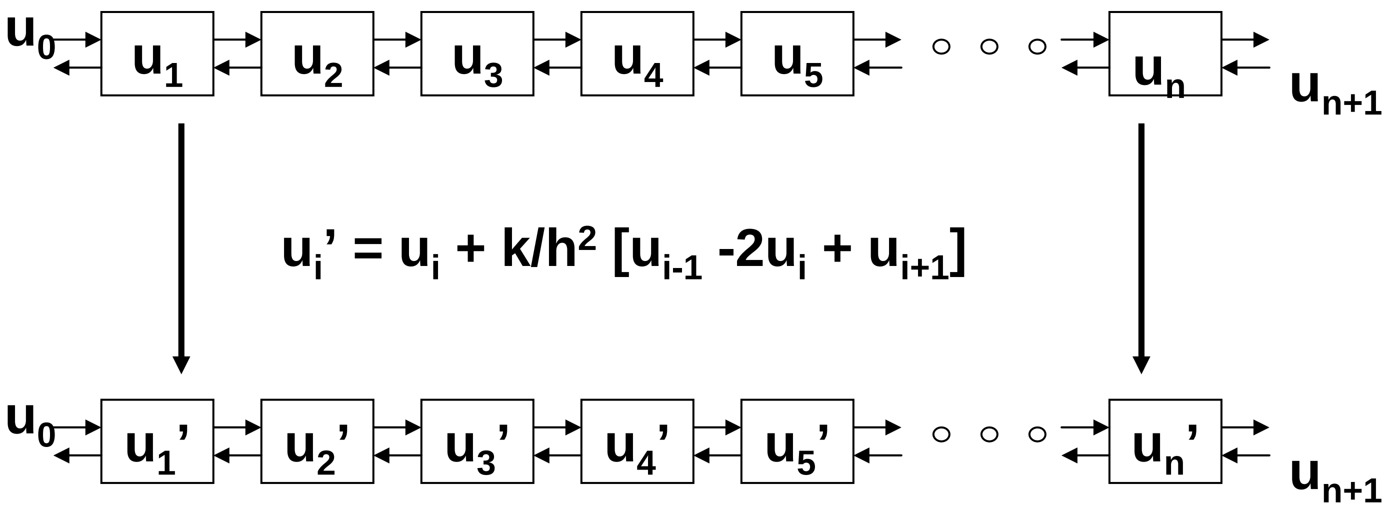


$$u_i' = u_i + k/h^2 \, [u_{i-1} - 2u_i + u_{i+1}]$$

Example: The Heat Equation (cont'd)

The following function computes the vector at the next timestep:

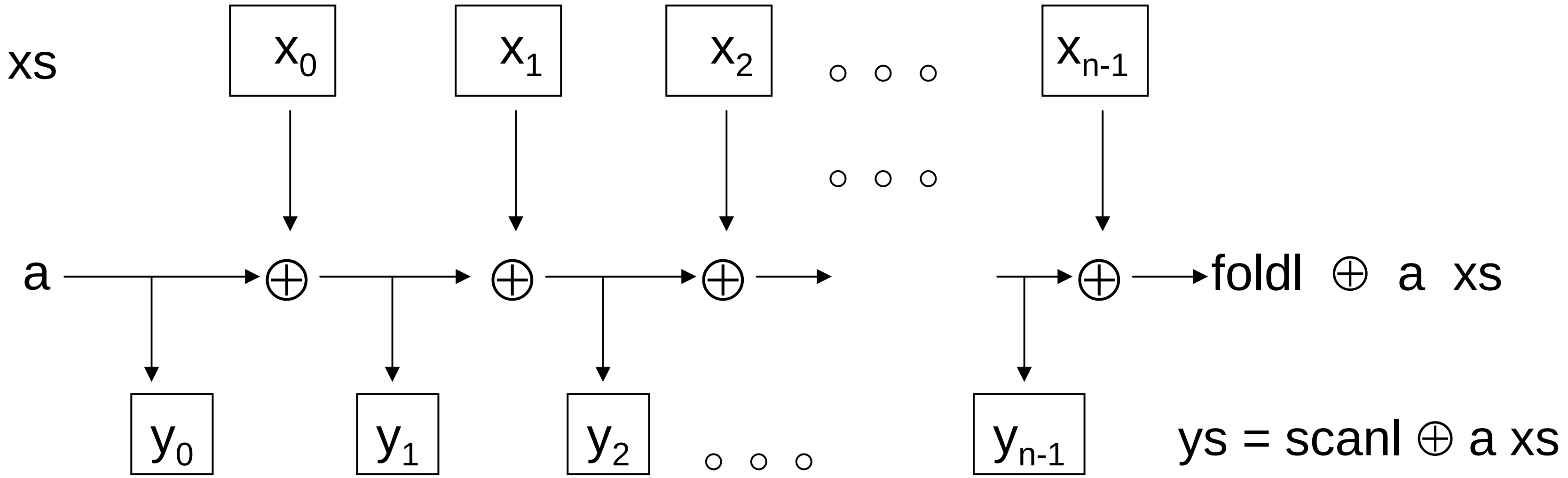
```
step               :: Float -> Float -> [Float] -> [Float]
step u0 un+1 us    = map g (zip us zs)
                     where
                       g (x, (a,b)) = (k / h*h) * (a - 2*x + b)
                       (a',b',zs) = shift u0 un+1 (map (\ u -> (u,u)) us)
```



Reduction Combinators

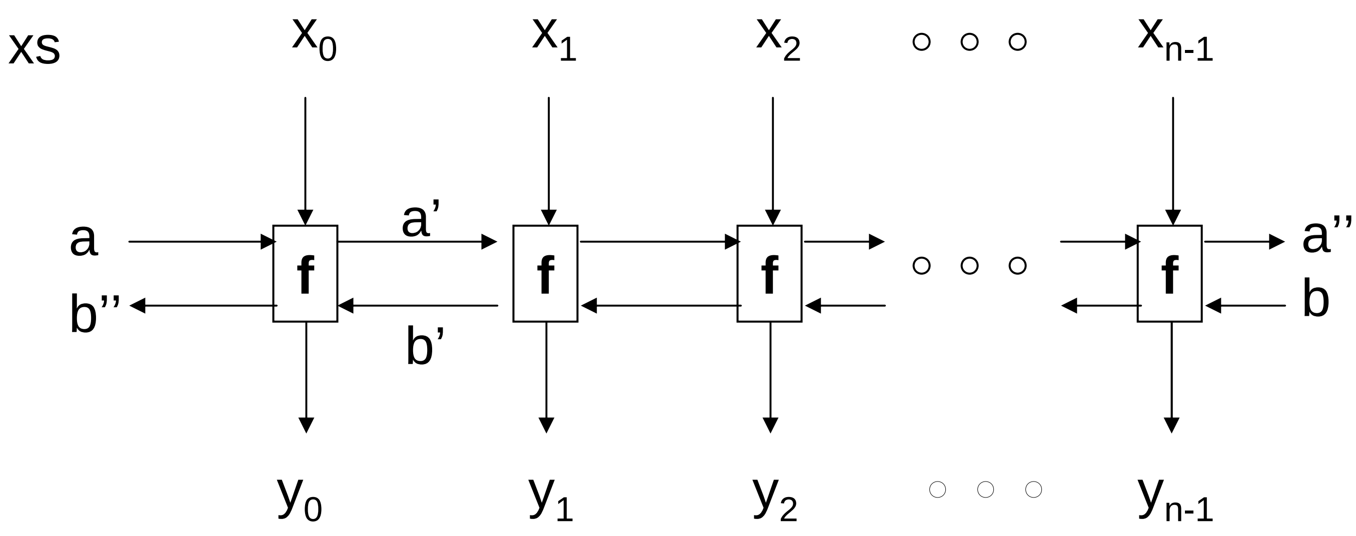
- **Combine Computation with Communication**
- **folding:** **foldl :: (a -> b -> a) -> a -> [b] -> a**
  foldl f a [ ] = a
  foldl f a (x:xs) = foldl f (f a x) xs

> **only seen as specification of the semantics, not as an implementation**



- **scanning:** **scanl :: (a -> b -> a) -> a -> [b] -> [a]**
  scanl f a xs = [foldl f a (take i xs) | i <- [0..length xs-1]]

Bidirectional Map-Scan




```
mscan :: (a -> b -> c -> (a,b,d)) -> a -> b -> [c] -> (a,b,[d])
mscan f a b [ ]        = (a, b, [ ])
mscan f a b (x:xs)     = (a'', b'', x' : xs')
                         where (a'', b', xs') = mscan f a' b xs
                               (a', b'', x')  = f a b' x
```

Example: Maximum Segment Sum

- **Problem: Take a list of numbers, and find the largest possible sum over any segment of contiguous numbers within the list.**

- **Example: [-500, 3, 4, 5, 6, -9, -8, 10, 20, 30, -9, 1, 2]**

  segment with maximum sum

- **Solution: For each i, where $0 \leq i < n$, let p_i be the maximum segment sum which is constrained to contain x_i, and let ps be the list of all the p_i.**

  Then the maximum segment sum for the entire list is just fold max ps.

  How can be compute the maximum segment sum which is constrained to contain x_i?

Example: Maximum Segment Sum

- **The following function returns the list of maximum segment sums for each element as well as the overall result:**

```
mss :: [Int] -> (Int, [Int])
mss xs     = (fold max ps, ps)
             where
                 (a', b', ps) = mscan g 0 0 xs
                 g  a  b  x  = (max 0 (a+x), max 0 (b+x), a + b + x)
```

- **Examples:**

mss [-500, 1, 2, 3, -500, 4, 5, 6, -500]
=>   (15, [-494, 6, 6, 6, -479, 15, 15, 15, -485])

mss [-500, 3, 4, 5, 6, -9, -8, 10, 20, 30, -9, 1, 2]
=>   (61, (-439, 61, 61, 61, 61, 61, 61, 61, 61, 61, 54, 54, 54])

Summary

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

Conclusions and Outlook

- **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 ......