Faculty of Science

Eden: Parallel Processes, Patterns and Skeletons

Jost Berthold

berthold@diku.dk
Department of Computer Science

Contents

Contents

Learning Goals:

- Writing programs in the parallel Haskell dialect Eden
- Reasoning about the behaviour of Eden programs.
- Applying and implementing parallel skeletons in Eden

Parallel Dialects of Haskell

- Data-Parallel Haskell[‡] (pure)
  Type-driven parallel operations (on parallel arrays), sophisticated compilation (vectorisation, fusion, . . .)
- Glasgow Parallel Haskell[‡,*] (pure)
  `par`, `seq` annotations for evaluation control, Evaluation Strategies

Parallel Dialects of Haskell

- Data-Parallel Haskell[‡] (pure)
  Type-driven parallel operations (on parallel arrays), sophisticated compilation (vectorisation, fusion, . . .)
- Glasgow Parallel Haskell[‡,*] (pure)
  `par, seq` annotations for evaluation control, Evaluation Strategies
- Eden[*] ("pragmatically impure")
  explicit process notion (mostly functional semantics), Distributed Memory (per process), implicit/explicit message passing

‡: shared memory, ∗: distributed memory

Parallel Dialects of Haskell

- Data-Parallel Haskell[‡] (pure)
  Type-driven parallel operations (on parallel arrays), sophisticated compilation (vectorisation, fusion, . . .)
- Glasgow Parallel Haskell[‡,*] (pure)
  `par`, `seq` annotations for evaluation control, Evaluation Strategies
- Eden[*] ("pragmatically impure")
  explicit process notion (mostly functional semantics), Distributed Memory (per process), implicit/explicit message passing
- Concurrent Haskell[‡], Eden implementation[*] (monadic)
  explicit thread control and communication, full programmer control and responsibility
- Par Monad, Cloud Haskell (monadic)
  newer explicit variants, approach similar to Eden implementation

‡: shared memory, ∗: distributed memory

Eden Constructs in a Nutshell

- Developed since 1996 in Marburg and Madrid
- Haskell, extended by communicating processes for coordination

Eden Constructs in a Nutshell

- Developed since 1996 in Marburg and Madrid
- Haskell, extended by communicating processes for coordination

Eden constructs for Process abstraction and instantiation

```
process ::(Trans a, Trans b)=> (a -> b) -> Process a b
( # ) :: (Trans a, Trans b) => (Process a b) -> a -> b
spawn :: (Trans a, Trans b) => [ Process a b ] -> [a] -> [b]
```

- Distributed Memory (Processes do not share data)
- Data sent through (hidden) 1:1 channels
- Type class `Trans`:
  - stream communication for lists
  - concurrent evaluation of tuple components
- Full evaluation of process output (if any result demanded)
- Non-functional features: explicit communication, $n:1$ channels

Quick Sidestep: WHNF, NFData and Evaluation

- Weak Head Normal Form (WHNF):
  Evaluation up to the top level constructor

Quick Sidestep: WHNF, NFData and Evaluation

- Weak Head Normal Form (WHNF):
  Evaluation up to the top level constructor

- Normal Form (NF):
  Full evaluation (recursively in sub-structures)

From *Control.DeepSeq*

```
class NFData a where
    rnf :: a -> ()      -- This was a _Strategy_ in 1998
    rnf a = a `seq` ()  -- returning unit ()

instance NFData Int
instance NFData Double
...
instance (NFData a) => NFData [a] where
    rnf [] = ()
    rnf (x:xs) = rnf x `seq` rnf xs
...
instance (NFData a, NFData b) => NFData (a,b) where
    rnf (a,b) = rnf a `seq` rnf b
```

Essential Eden: Process Abstraction/Instantiation

Process **Abstraction**: `process ::... (a -> b) -> Process a b`

```
multproc = process (\x -> [ x*k | k <- [1,2..]])
```

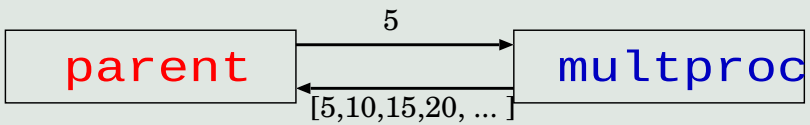
Essential Eden: Process Abstraction/Instantiation

Process **Abstraction**: `process ::... (a -> b) -> Process a b`

```
multproc = process (\x -> [ x*k | k <- [1,2..]])
```

Process **Instantiation**: `(#) ::... Process a b -> a -> b`

```
multiple5 = multproc # 5
```



- Full evaluation of argument (concurrent) and result (parallel)
- Stream communication for lists

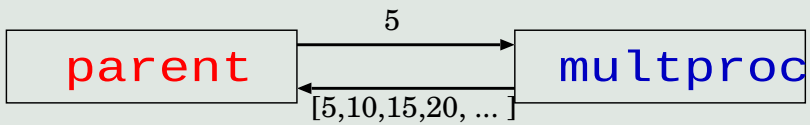
Essential Eden: Process Abstraction/Instantiation

Process **Abstraction**: `process ::... (a -> b) -> Process a b`

```
multproc = process (\x -> [ x*k | k <- [1,2..]])
```

Process **Instantiation**: `(#) ::... Process a b -> a -> b`

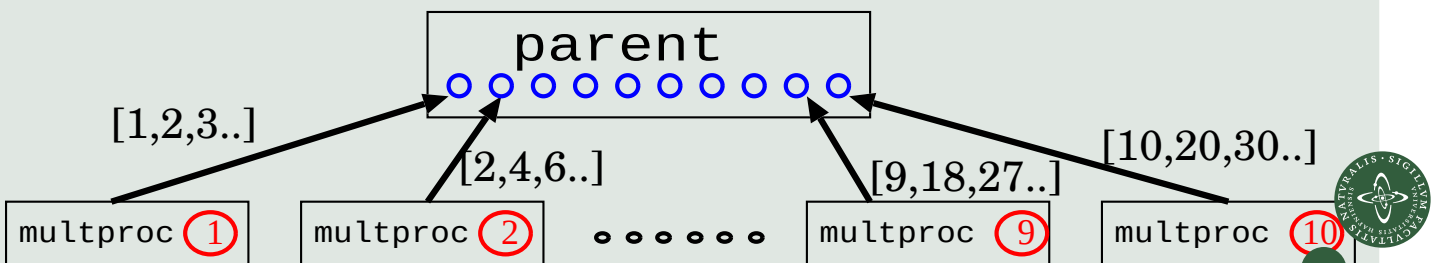
```
multiple5 = multproc # 5
```



- Full evaluation of argument (concurrent) and result (parallel)
- Stream communication for lists

Spawning **multiple processes**: `spawn ::... [Process a b] -> [a] -> [b]`

```
multiples = spawn (replicate 10 multproc) [1..10]
```



A Small Eden Example[1]

- Subexpressions evaluated in parallel
- . . . in different processes with separate heaps

simpleeden.hs

```
main = do args <- getArgs
          let first_stuff = (process f_expensive) # (args!!0)
              other_stuff = g_expensive $# (args!!1) -- syntax variant
          putStrLn (show first_stuff ++ '\n':show other_stuff)
```

[1] (compiled with option -parcp or -parmpi)

A Small Eden Example[1]

- Subexpressions evaluated in parallel
- . . . in different processes with separate heaps

simpleeden.hs

```
main = do args <- getArgs
          let first_stuff = (process f_expensive) # (args!!0)
              other_stuff = g_expensive $# (args!!1) -- syntax variant
          putStrLn (show first_stuff ++ '\n':show other_stuff)
```

. . . which will not produce any speedup!

[1] (compiled with option -parcp or -parmpi)

A Small Eden Example[1]

- Subexpressions evaluated in parallel
- . . . in different processes with separate heaps

simpleeden.hs

```
main = do args <- getArgs
          let first_stuff = (process f_expensive) # (args!!0)
              other_stuff = g_expensive $# (args!!1) -- syntax variant
          putStrLn (show first_stuff ++ '\n':show other_stuff)
```

. . . which will not produce any speedup!

simpleeden2.hs

```
main = do args <- getArgs
          let [first_stuff,other_stuff]
                = spawnF [f_expensive, g_expensive] args
          putStrLn (show first_stuff ++ '\n':show other_stuff)
```

- Processes are created when there is demand for the result!
- Spawn both processes at the same time using special function.

[1] (compiled with option -parcp or -parmpi)

Basic Eden Exercise: Hamming Numbers

The Hamming Numbers are defined as the ascending sequence of numbers:

$$\left\{2^i \cdot 3^j \cdot 5^k \mid i, j, k \in \mathbb{N}\right\}$$

Basic Eden Exercise: Hamming Numbers

The Hamming Numbers are defined as the ascending sequence of numbers:

$$\left\{2^i \cdot 3^j \cdot 5^k \mid i, j, k \in \mathbb{N}\right\}$$

Dijkstra:

The first Hamming number is 1. Each following Hamming number H can be written as $H = 2K$, $H = 3K$, or $H = 5K$; with a suitable smaller Hamming number K.

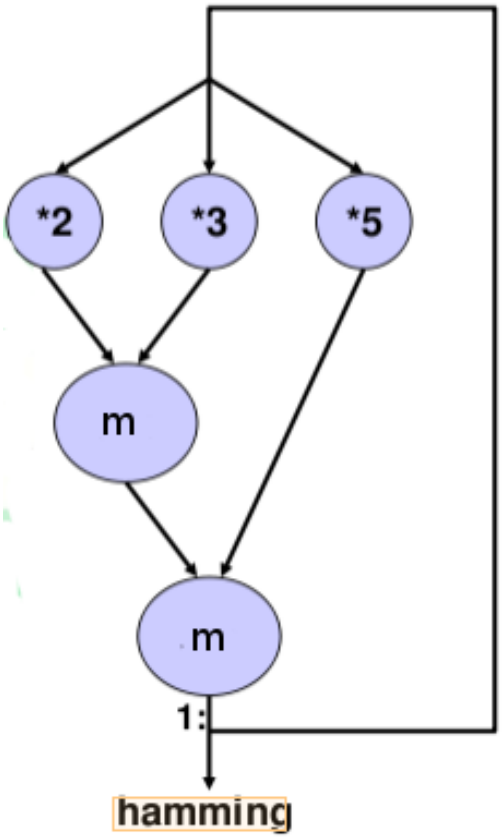
Basic Eden Exercise: Hamming Numbers

The Hamming Numbers are defined as the ascending sequence of numbers:

$$\left\{2^i \cdot 3^j \cdot 5^k \mid i, j, k \in \mathbb{N}\right\}$$

Dijkstra:
The first Hamming number is 1. Each following Hamming number H can be written as $H = 2K$, $H = 3K$, or $H = 5K$; with a suitable smaller Hamming number K.

- Write an Eden program that produces Hamming numbers using parallel processes. The program should take one argument n and produce the numbers up to position n.
- Observe the parallel behaviour of your program using EdenTV.



Non-Functional Eden Constructs for Optimisation

Location-Awareness: `noPe, selfPe :: Int`

```
spawnAt :: (Trans a, Trans b) => [Int] -> [Process a b] -> [a] -> [b]
instantiateAt :: (Trans a, Trans b) =>
                 Int -> Process a b -> a -> IO b
```

Non-Functional Eden Constructs for Optimisation

Location-Awareness: `noPe, selfPe :: Int`

```
spawnAt :: (Trans a, Trans b) => [Int] -> [Process a b] -> [a] -> [b]
instantiateAt :: (Trans a, Trans b) =>
                 Int -> Process a b -> a -> IO b
```

Explicit communication using primitive operations (monadic)

```
data ChanName = Comm (Channel a -> a -> IO ())
createC :: IO (Channel a , a)

class NFData a => Trans a where
    write :: a -> IO ()
    write x = rdeepseq x `pseq` sendData Data x
    createComm :: IO (ChanName a, a)
    createComm = do (cx,x) <- createC
                    return (Comm (sendVia cx) , x)
```

Nondeterminism! `merge :: [[a]] -> [a]`

Hidden inside a Haskell module, only for the library implementation.

Outline

The Idea of Skeleton-Basked Parallelism

You have already seen one example:

- (Binary) Divide and Conquer, as a higher-order function

```
divConqB :: (a -> b) -> a      -- base case fct., input
            -> (a -> Bool)     -- parallel threshold
            -> (b -> b -> b)   -- combine
            -> (a -> Maybe (a,a)) -- divide
            -> b
divConqB baseF input doSeq combine divide = ...
```

(and another version, explained more lated)

- Parallel structure (binary tree) exploited for parallelism
- Abstracted from concrete problem

The Idea of Skeleton-Basked Parallelism

You have already seen one example:

- (Binary) Divide and Conquer, as a higher-order function

```
divConqB :: (a -> b) -> a      -- base case fct., input
            -> (a -> Bool)     -- parallel threshold
            -> (b -> b -> b)   -- combine
            -> (a -> Maybe (a,a)) -- divide
            -> b
divConqB baseF input doSeq combine divide = ...
```

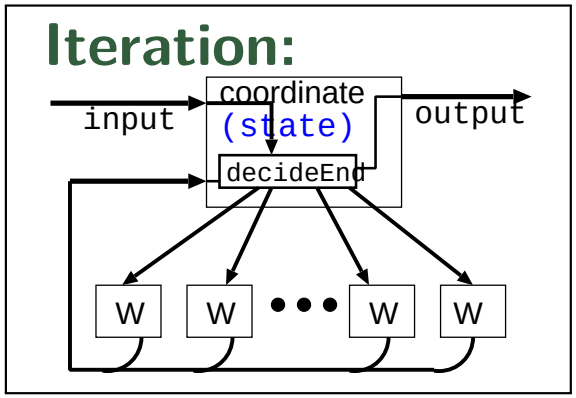
(and another version, explained more lated)

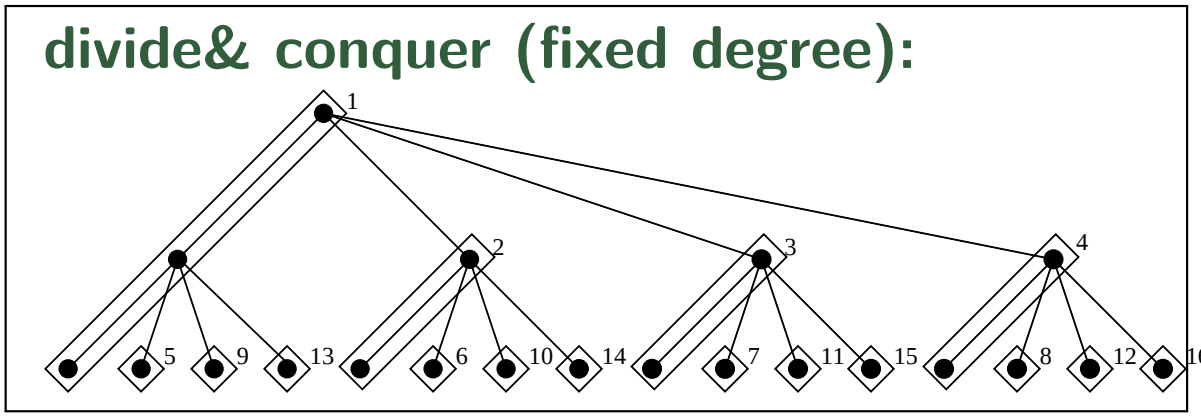
- Parallel structure (binary tree) exploited for parallelism
- Abstracted from concrete problem

And another one, much simpler, much more common:

```
parMap ::  (a->b) -> [a] -> [b]
```

Algorithmic Skeletons for Parallel Programming

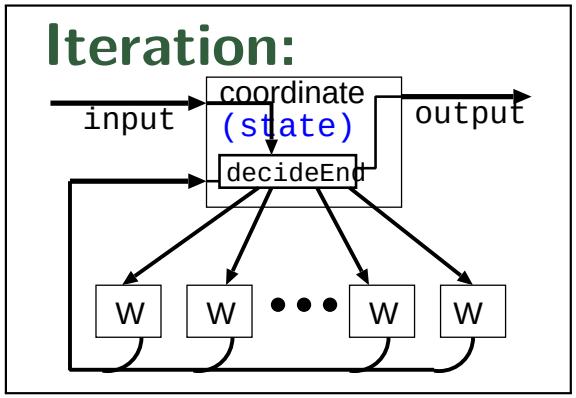


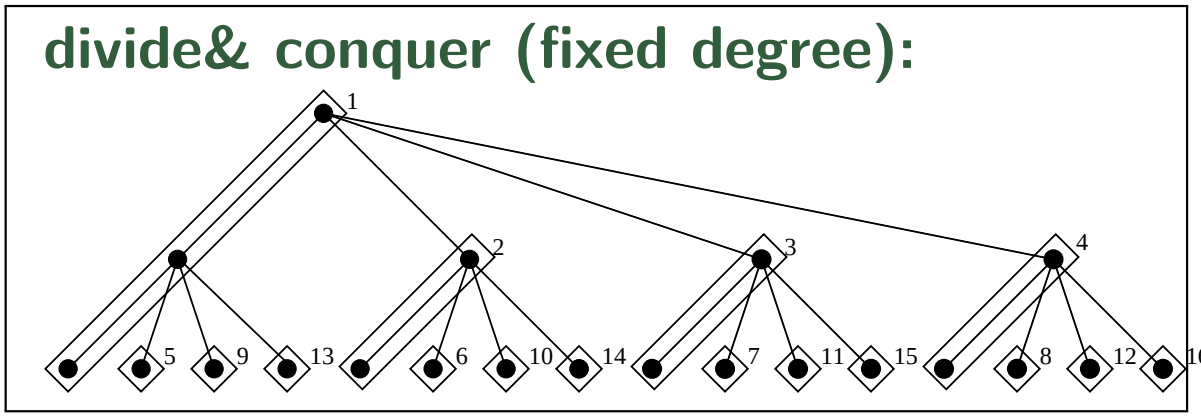


Algorithmic Skeletons [Cole 1989]: Boxes and lines – executable!

- Abstraction of algorithmic structure as a higher-order function

Algorithmic Skeletons for Parallel Programming

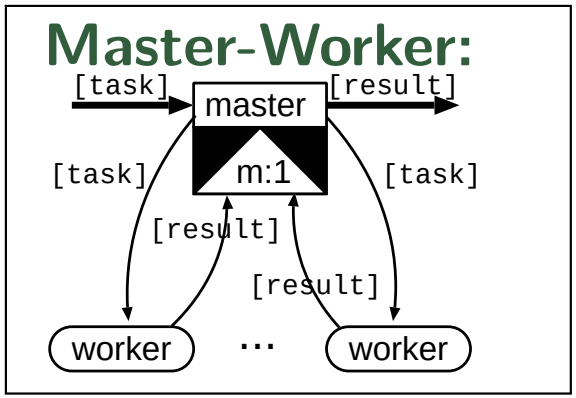


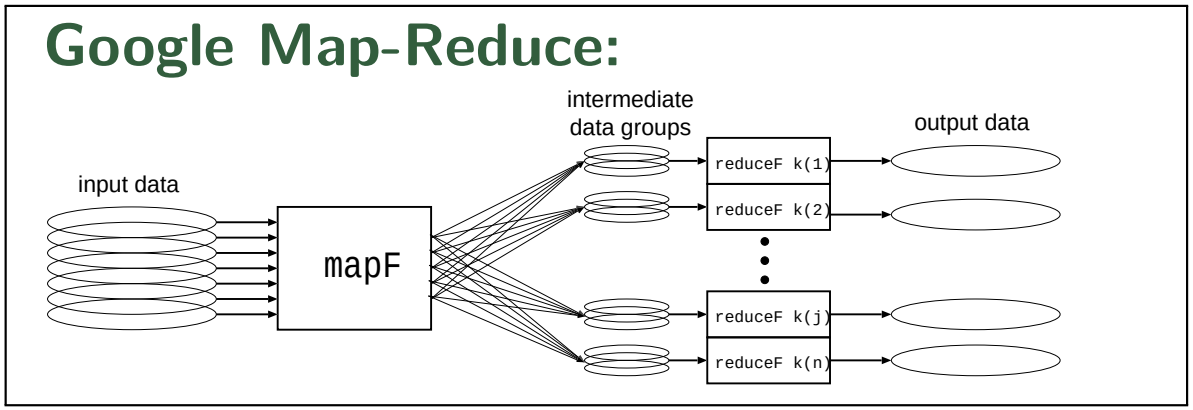


Algorithmic Skeletons [Cole 1989]: Boxes and lines – executable!

- Abstraction of algorithmic structure as a higher-order function
- Embedded "worker" functions (by application programmer)
- Hidden parallel library implementation (by system programmer)

Algorithmic Skeletons for Parallel Programming

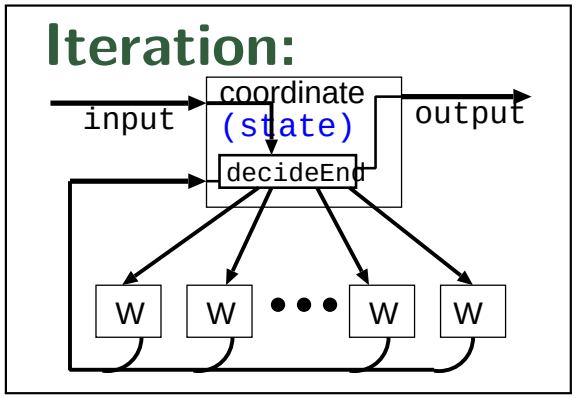


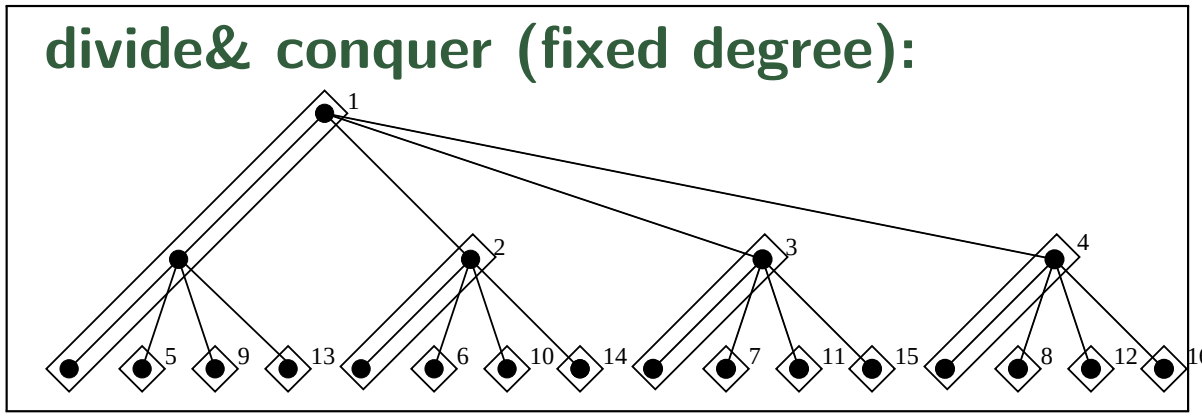


Algorithmic Skeletons [Cole 1989]: Boxes and lines – executable!

- Abstraction of algorithmic structure as a higher-order function
- Embedded "worker" functions (by application programmer)
- Hidden parallel library implementation (by system programmer)
- Different kinds of skeletons: topological, small-scale, algorithmic

Algorithmic Skeletons for Parallel Programming





Algorithmic Skeletons [Cole 1989]: Boxes and lines – executable!

- Abstraction of algorithmic structure as a higher-order function
- Embedded "worker" functions (by application programmer)
- Hidden parallel library implementation (by system programmer)
- Different kinds of skeletons: topological, small-scale, algorithmic

Explicit parallelism control and functional paradigm are a good setting to implement and use skeletons for parallel programming.

Types of Skeletons

Common Small-scale Skeletons

- encapsulate common parallelisable operations or patterns
- parallel behaviour (concrete parallelisation) hidden

Structure-oriented: Topology Skeletons

- describe interaction between execution units
- explicitly model parallelism

Proper Algorithmic Skeletons

- capture a more complex algorithm-specific structure
- sometimes domain-specific

Outline

Basic Skeletons: Higher-Order Functions

- Parallel transformation: Map

```
map :: (a -> b) -> [a] -> [b]
```

independent elementwise transformation

. . . probably the most common example of parallel functional programming (called "embarassingly parallel")

Basic Skeletons: Higher-Order Functions

- Parallel transformation: Map

```
map :: (a -> b) -> [a] -> [b]
```

  independent elementwise transformation

  . . . probably the most common example of parallel functional programming (called "embarassingly parallel")

- Parallel Reduction: Fold

```
fold :: (a -> a -> a) -> a -> [a] -> a
```

  with commutative and associative operation.

- Parallel (left) Scan:

```
parScanL :: (a -> a -> a) -> [a] -> [a]
```

  reduction keeping the intermediate results.

- Parallel Map-Reduce:

  combining transformation and groupwise reduction.

Embarassingly Parallel: `map`

`map`: apply transformation to all elements of a list

- Straight-forward element-wise parallelisation

```
parmap :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parmap  = spawn . repeat . process
     -- parmap f xs = spawn (repeat (process f)) xs
```

Embarassingly Parallel: `map`

`map`: apply transformation to all elements of a list

- Straight-forward element-wise parallelisation

```
parmap :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parmap  = spawn . repeat . process
     -- parmap f xs = spawn (repeat (process f)) xs
```

Much too fine-grained!

Embarassingly Parallel: map

map: apply transformation to all elements of a list

- Straight-forward element-wise parallelisation

```
parmap :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parmap  = spawn . repeat . process
     -- parmap f xs = spawn (repeat (process f)) xs
```

Much too fine-grained!

- Group-wise processing: Farm of processes

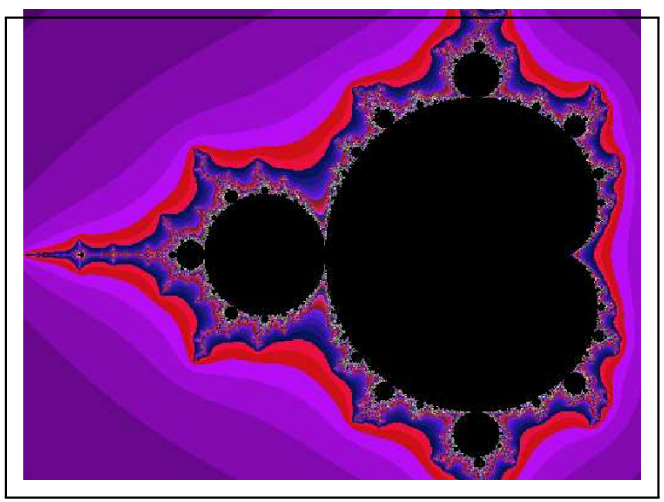
```
farm :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
farm f xs = join results
    where results = spawn (repeat (process (map f))) parts
          parts   = distribute noPe xs -- noPe, so use all nodes
          join    :: [[a]] -> [a]
          join    = ...
          distribute  :: Int -> [a] -> [[a]]
          distribute n = ... -- join . distribute n == id
```

Example

Mandelbrot set visualisation $z_{n+1} = z_n^2 + c$ for $c \in \mathbb{C}$

Mandelbrot (Pseudocode)

```
pic :: ..picture-parameters.. -> PPMAscii
pic threshold ul lr dimx np s = ppmheader ++ concat (parMap computeRow rows)
   where rows = ...dimx..ul..lr..
         parMap = ...np..s..
```

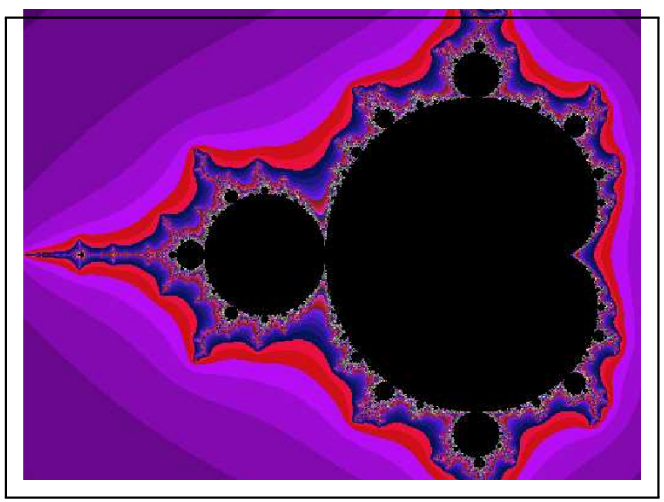


Example / Exercise

Mandelbrot set visualisation $z_{n+1} = z_n^2 + c$ for $c \in \mathbb{C}$

Mandelbrot (Pseudocode)

```
pic :: ..picture-parameters.. -> PPMAscii
pic threshold ul lr dimx np s = ppmheader ++ concat (parMap computeRow rows)
   where rows = ...dimx..ul..lr..
         parMap = ...np..s.. -- you define it
```



Exercise:

- Implement parMap in 2 different ways
- Run the Mandelbrot program with both versions, compare the behaviour.

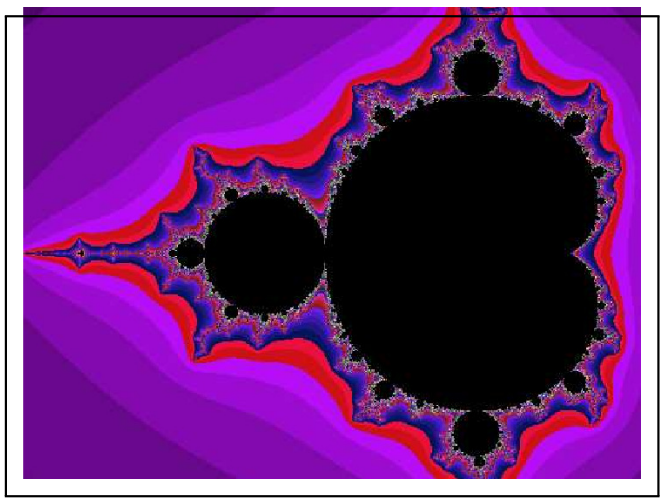
Framework programs can be found on the course pages...

Example / Exercise: Chunked Tasks

Mandelbrot set visualisation $z_{n+1} = z_n^2 + c$ for $c \in \mathbb{C}$

Mandelbrot (Pseudocode)

```
pic :: ..picture-parameters.. -> PPMAscii
pic threshold ul lr dimx np s = ppmheader ++ concat (parMap computeRow rows)
   where rows = ...dimx..ul..lr..
         parMap = ..using chunks..
```

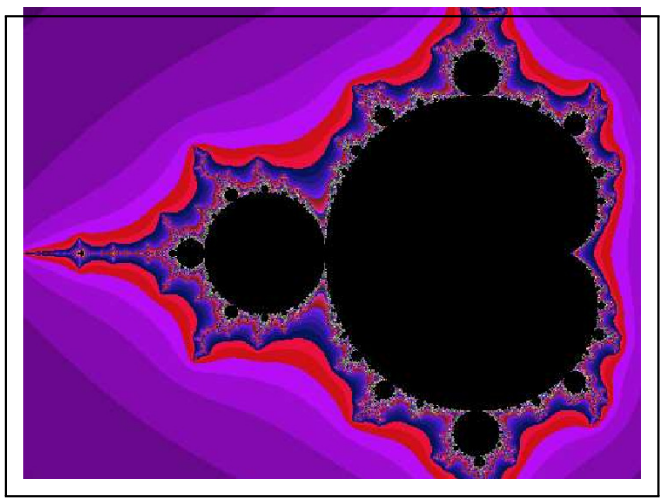


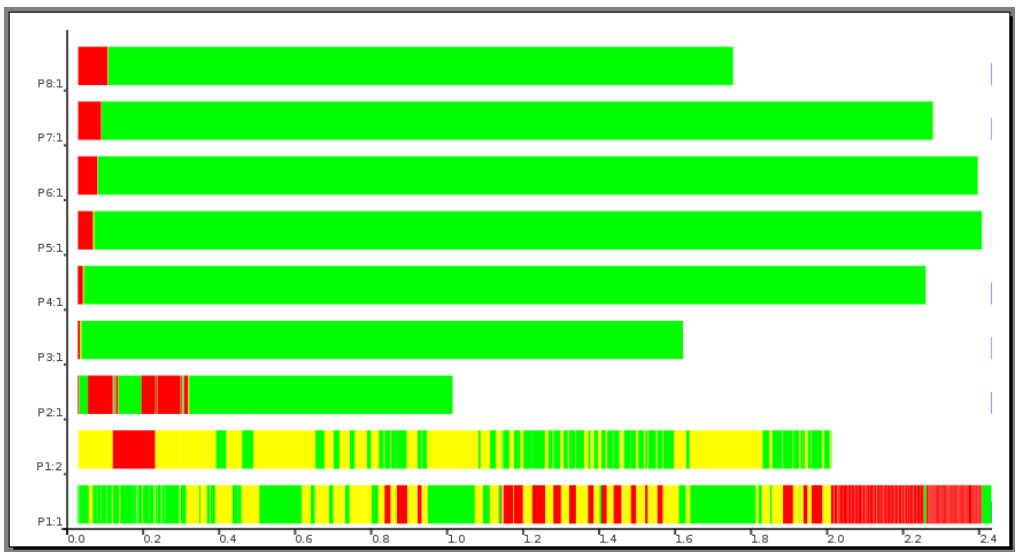
Example / Exercise: Chunked Tasks

Mandelbrot set visualisation $z_{n+1} = z_n^2 + c$ for $c \in \mathbb{C}$

Mandelbrot (Pseudocode)

```
pic :: ..picture-parameters.. -> PPMAscii
pic threshold ul lr dimx np s = ppmheader ++ concat (parMap computeRow rows)
   where rows = ...dimx..ul..lr..
         parMap = ..using chunks..
```





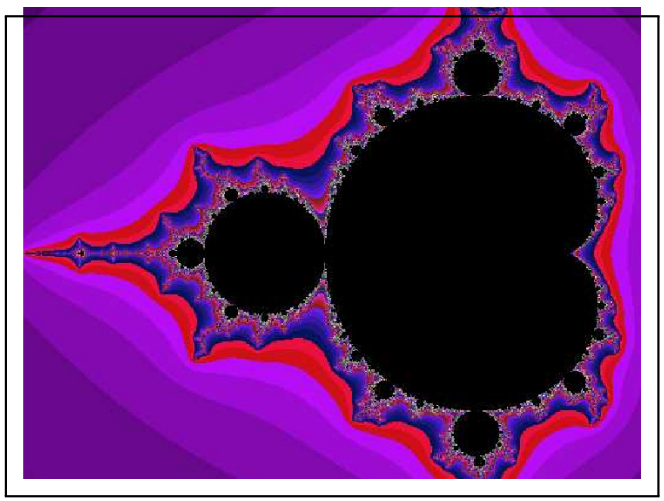
Simple chunking leads to load imbalance (task complexities differ)

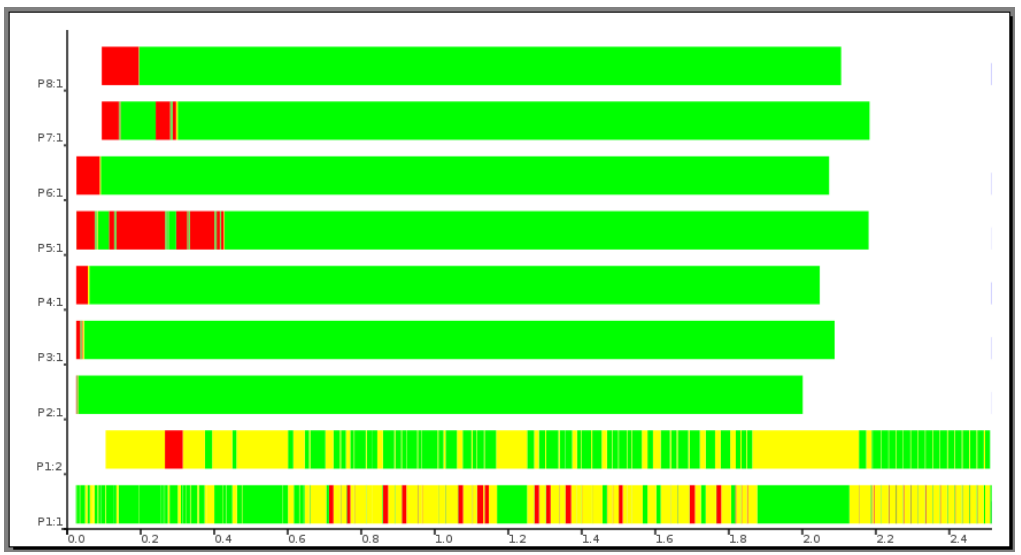
Example / Exercise: Round-robin Tasks

Mandelbrot set visualisation $z_{n+1} = z_n^2 + c$ for $c \in \mathbb{C}$

Mandelbrot (Pseudocode)

```
pic :: ..picture-parameters.. -> PPMAscii
pic threshold ul lr dimx np s = ppmheader ++ concat (parMap computeRow rows)
   where rows = ...dimx..ul..lr..
         parMap = ..distributing round-robin..
```





Better: round-robin distribution, but still not well-balanced.

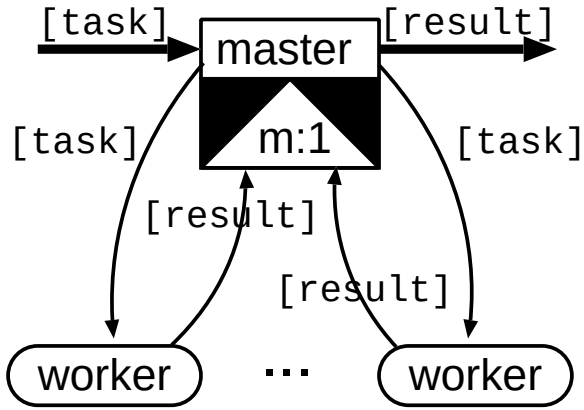
Master-Worker Skeleton

Worker nodes transform elementwise:

```
worker ::  task -> result
```

Master node manages task pool

```
mw  :: Int -> Int ->
       ( a -> b ) -> [a] -> [b]
mw np prefetch f tasks =  ...
```



Parameters: no. of workers, prefetch

- Master sends a new task each time a result is returned (needs many-to-one communication)
- Initial workload of `prefetch` tasks for each worker:
  Higher prefetch $\Rightarrow$ more and more static task distribution
  Lower prefetch $\Rightarrow$ dynamic load balance

Master-Worker Skeleton

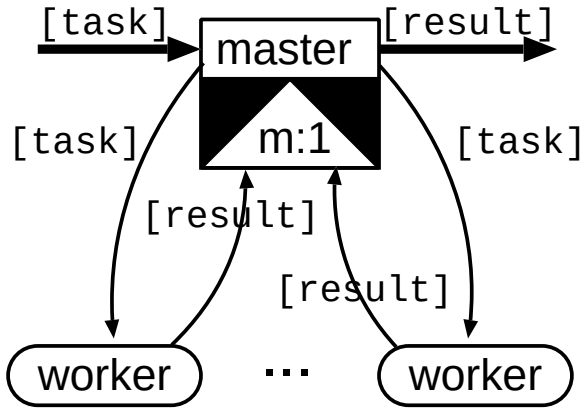
Worker nodes transform elementwise:

```
worker ::  task -> result
```

Master node manages task pool

```
mw  :: Int -> Int ->
       ( a -> b ) -> [a] -> [b]
mw np prefetch f tasks =  ...
```

Parameters: no. of workers, prefetch



- Master sends a new task each time a result is returned (needs many-to-one communication)
- Initial workload of `prefetch` tasks for each worker:
  Higher prefetch $\Rightarrow$ more and more static task distribution
  Lower prefetch $\Rightarrow$ dynamic load balance
- Result order needs to be reestablished!

Master-Worker: An Implementation

Master-Worker Skeleton Code

```
mw np prefetch f tasks = results
 where
  fromWorkers         = spawn workerProcs toWorkers
  workerProcs         = [process (zip [n,n..] . map f) | n<-[1..np]]
  toWorkers           = distribute tasks requests
```

- Workers tag results with their ID (between 1 and `np`).

Master-Worker: An Implementation

Master-Worker Skeleton Code

```
mw np prefetch f tasks = results
 where
  fromWorkers         = spawn workerProcs toWorkers
  workerProcs         = [process (zip [n,n..] . map f) | n<-[1..np]]
  toWorkers           = distribute tasks requests

  (newReqs, results) = (unzip . merge) fromWorkers
  requests            = initialReqs ++ newReqs
  initialReqs         = concat (replicate prefetch [1..np])
```

- Workers tag results with their ID (between 1 and `np`).
- Result streams are non-deterministically merged into one stream.

Master-Worker: An Implementation

Master-Worker Skeleton Code

```
mw np prefetch f tasks = results
 where
  fromWorkers         = spawn workerProcs toWorkers
  workerProcs         = [process (zip [n,n..] . map f) | n<-[1..np]]
  toWorkers           = distribute tasks requests

  (newReqs, results) = (unzip . merge) fromWorkers
  requests            = initialReqs ++ newReqs
  initialReqs         = concat (replicate prefetch [1..np])

  distribute :: [t] -> [Int] -> [[t]]
  distribute tasks reqs = [taskList reqs tasks n | n<-[1..np]]
     where taskList (r:rs) (t:ts) pe | pe == r    = t:(taskList rs ts pe)
                                     | otherwise  =    taskList rs ts pe
           taskList _      _      _  = []
```

- Workers tag results with their ID (between 1 and `np`).
- Result streams are non-deterministically merged into one stream.
- The `distribute` function supplies new tasks according to requests.

Parallel Reduction, Map-Reduce

Reduction (`fold`) usually has a direction

- ```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldr :: (a -> b -> b) -> b -> [a] -> b
  ```

  Starting from the left or right, implying different reduction function.
- To parallelise: break into sublists and pre-reduce in parallel.
- Better options if order does not matter.


  ```

Parallel Reduction, Map-Reduce

Reduction (`fold`) usually has a direction

- ```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldr :: (a -> b -> b) -> b -> [a] -> b
  ```

  Starting from the left or right, implying different reduction function.
- To parallelise: break into sublists and pre-reduce in parallel.
- Better options if order does not matter.

Example: $\sum_{k=1}^{n} \varphi(k) = \sum_{k=1}^{n} |\{j < k \mid gcd(k,j) = 1\}|$ (Euler Phi)

sumEuler

```
result = foldl (+) 0 (map phi [1..n])
phi k = length (filter (\ n -> gcd n k == 1) [1..(k-1)])
```


```

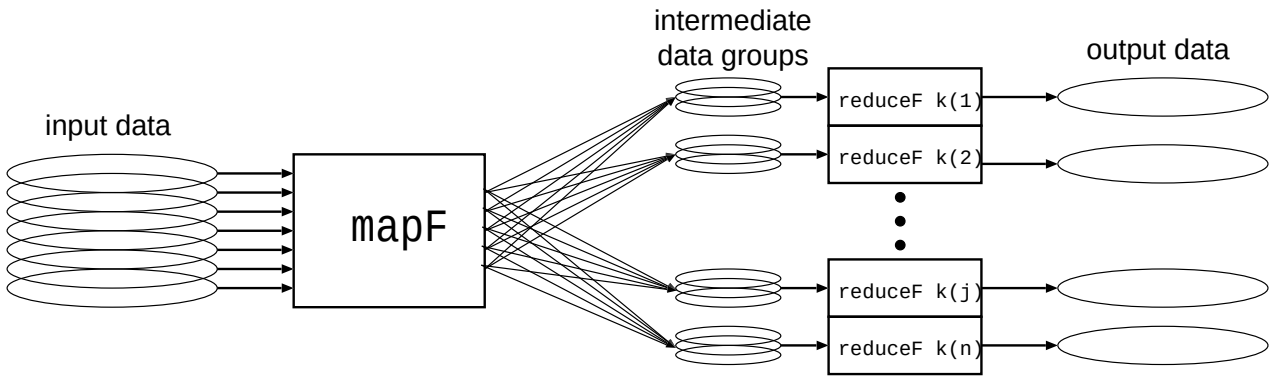
Parallel Map-Reduce: Restrictions

- ```
  parmapReduceStream :: Int ->
                        (a -> b) -> (b -> b -> b) -> b ->
                        [a] -> b
  parmapReduceStream np mapF redF neutral list = foldl redF neutral subRs
      where sublists = distribute np list
            subFold  = process (foldl' redF neutral . (map mapF))
            subRs    = spawn (replicate np subFold) sublists
  ```


  ```

Parallel Map-Reduce: Restrictions

- ```
  parmapReduceStream :: Int ->
                        (a -> b) -> (b -> b -> b) -> b ->
                        [a] -> b
  parmapReduceStream np mapF redF neutral list = foldl redF neutral subRs
      where sublists = distribute np list
            subFold  = process (foldl' redF neutral . (map mapF))
            subRs    = spawn (replicate np subFold) sublists
  ```
- Associativity and neutral element (essential).
- commutativity (desired, more liberal distribution)
- need to narrow type of the reduce parameter function!
- . . . Alternative `fold` type:
  ```
  redF' :: [b] -> b
  redF'   []   = neutral
  redF' (x:xs) = foldl' redF x xs
  ```


  ```

Google Map-Reduce: Grouping Before Reduction

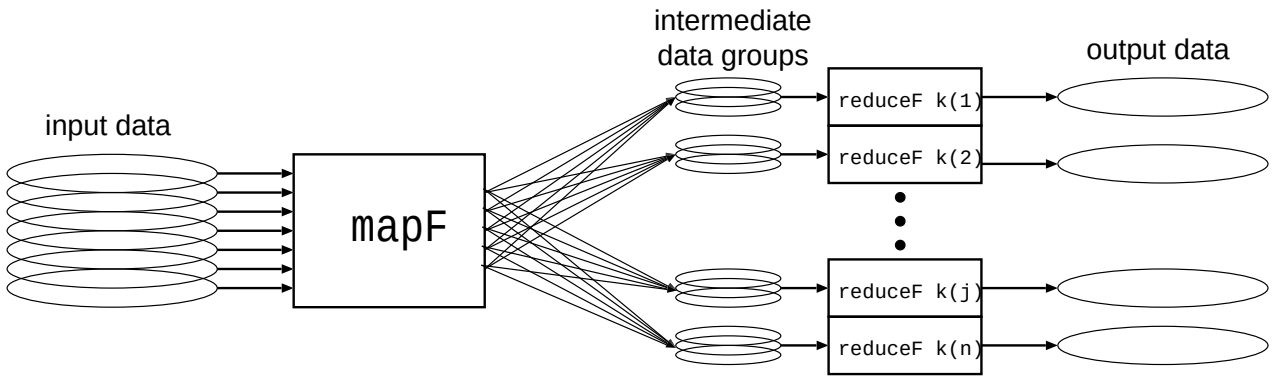
```
gMapRed :: (k1 -> v1 -> [(k2,v2)])      -- mapF
        -> (k2 -> [v2] -> Maybe v3) -- reduceF
        -> Map k1 v1 -> Map k2 v3   -- input / output
```



1. Input: key-value pairs `(k1,v1)`, many or no outputs `(k2,v2)`
2. Intermediate grouping by key `k2`
3. Reduction per (intermediate) key `k2` (maybe without result)
4. Input and output: Finite mappings

Google Map-Reduce: Grouping Before Reduction

```
gMapRed :: (k1 -> v1 -> [(k2,v2)])      -- mapF
        -> (k2 -> [v2] -> Maybe v3) -- reduceF
        -> Map k1 v1 -> Map k2 v3   -- input / output
```

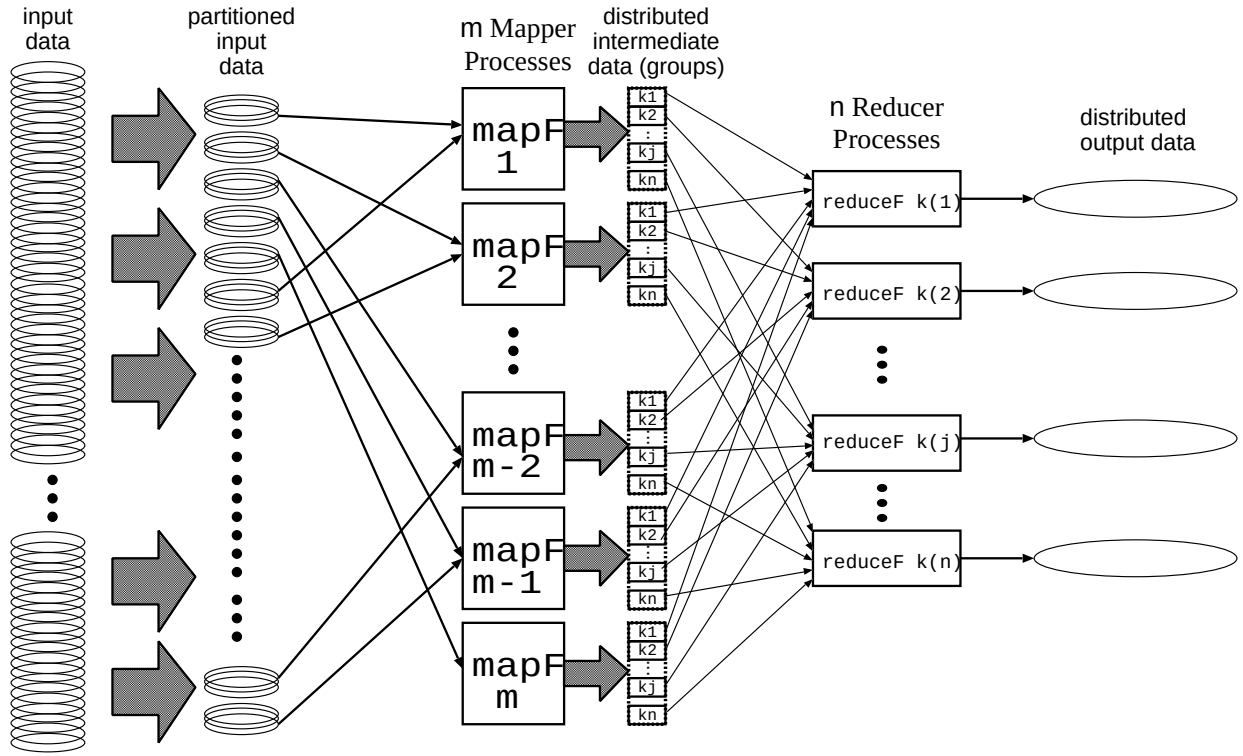



```
(uRL,document)    ==>    [(word,1)]  ==>   (word :-> count)
```

Word Occurrence

```
mapF :: URL -> String -> [(String,Int)]
mapF _ content = [(word,1) | word <- words content ]
reduceF :: String -> [Int] -> Maybe Int
reduceF word counts = Just (sum counts)
```

Google Map-Reduce (parallel)



R.Lämmel, Google's Map-Reduce Programming Model Revisited. In: SCP 2008

```
gMapRed :: Int -> (k2->Int) -> Int -> (v1->Int) -- parameters
           (k1 -> v1 -> [(k2,v2)])      -- mapper
           -> (k2 -> [v2] -> Maybe v3) -- pre-reducer
           -> (k2 -> [v3] -> Maybe v4) -- final reducer
           -> Map k1 v1 -> Map k2 v4   -- input / output
```

Examples / Exercise

```
gMapRed :: Int -> (k2->Int) -> Int -> (v1->Int) -- parameters
           (k1 -> v1 -> [(k2,v2)])      -- mapper
           -> (k2 -> [v2] -> Maybe v3) -- pre-reducer
           -> (k2 -> [v3] -> Maybe v4) -- final reducer
           -> [(k1,v1)] -> Map k2 v4   -- input / output
```

Describe how to compute the following in Google Map-Reduce:

- Reverse Web-Link Graph:
  For a set of URLs, compute a dictionary to enable looking up all pages that link to one particular page.

- URL Access Frequencies:
  Compute the access count for all URLs in a web server log file.

Examples / Exercise

```
gMapRed :: Int -> (k2->Int) -> Int -> (v1->Int) -- parameters
           (k1 -> v1 -> [(k2,v2)])      -- mapper
           -> (k2 -> [v2] -> Maybe v3) -- pre-reducer
           -> (k2 -> [v3] -> Maybe v4) -- final reducer
           -> [(k1,v1)] -> Map k2 v4   -- input / output
```

Describe how to compute the following in Google Map-Reduce:

- Reverse Web-Link Graph:
  For a set of URLs, compute a dictionary to enable looking up all pages that link to one particular page.

Reverse Link

Input are all URLs and page contents of the set. The map function outputs pairs (link target, source URL) for each link found in the source URL contents. The (pre-)reduce function joins the source URLs to the pair (target, list(source)) (removing duplicates).

- URL Access Frequencies:
  Compute the access count for all URLs in a web server log file.

URL Access Frequency

Input are all log entries, stating the requested URLs. As in word-occurrence: The map function emits (URL,1) pairs for requested URLs, the reduce functions sum the counts.

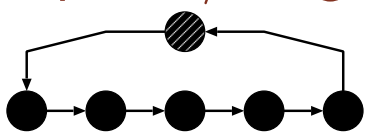
Outline

Process Topologies as Skeletons: Explicit Parallelism

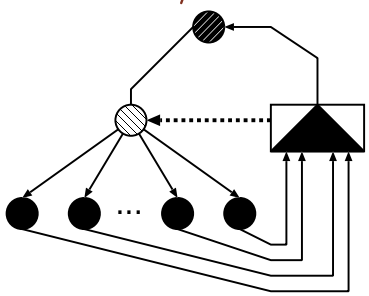
- describe typical patterns of parallel interaction structure
- (where node behaviour is the function argument)
- to structure parallel computations

Examples:

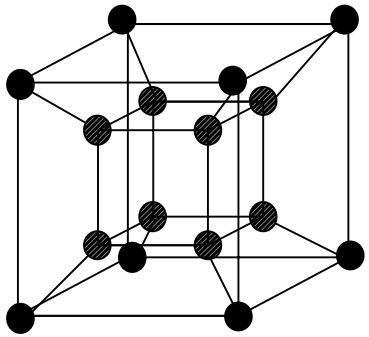
Pipeline/Ring:



Master/Worker:



Hypercube:

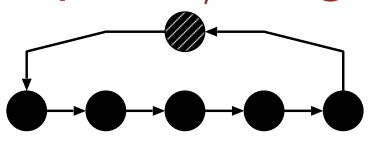


Process Topologies as Skeletons: Explicit Parallelism

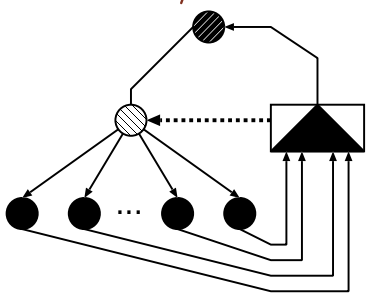
- describe typical patterns of parallel interaction structure
- (where node behaviour is the function argument)
- to structure parallel computations

Examples:

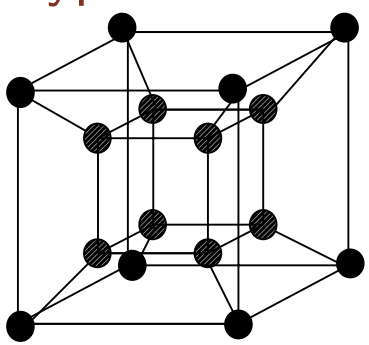
Pipeline/Ring:



Master/Worker:

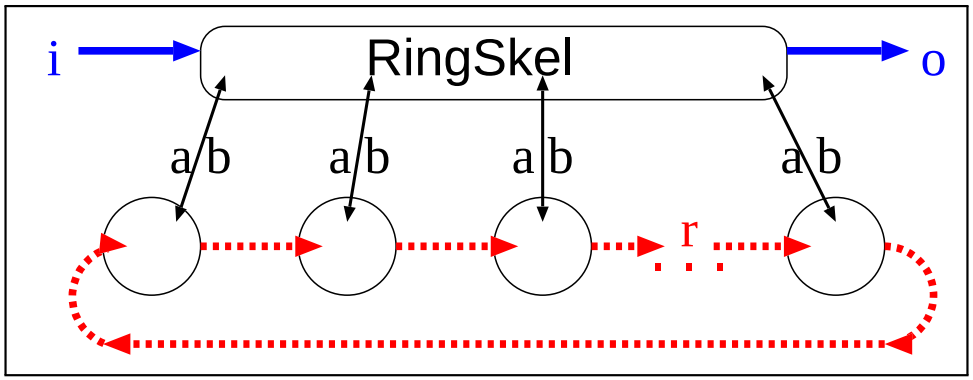


Hypercube:



$\Rightarrow$ well-suited for functional languages (with explicit parallelism).
Skeletons can be implemented and applied in Eden.

Process Topologies as Skeletons: Ring

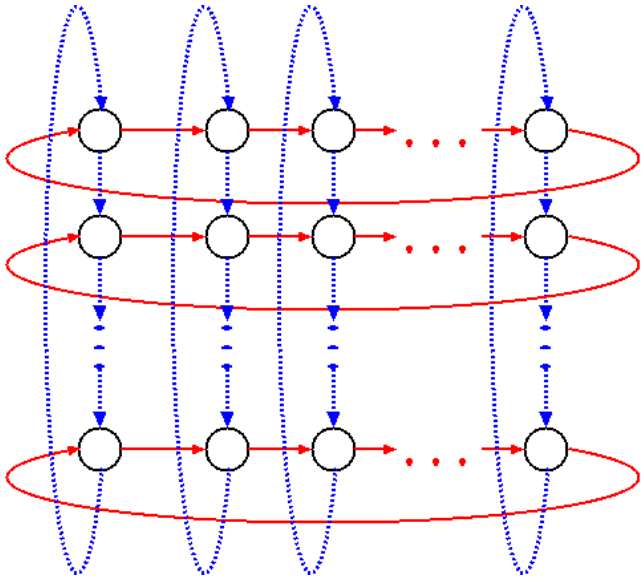



```
type RingSkel i o a b r = Int -> (Int -> i -> [a]) -> ([b] -> o) ->
                          ((a,[r]) -> (b,[r])) -> i -> o

ring size makeInput processOutput ringWorker input = ...
```

- Good for exchanging (updated) global data between nodes
- All ring processes connect to parent to receive input/send output
- Parameters: functions for
  - decomposing input, combining output, ring worker

Process Topologies as Skeletons: Torus

```
torus ::
   -- node behaviour
  (c->[a]->[b] -> (d,[a],[b])) ->
   -- input (truncated to shortest)
  [[c]] -> [[d]] -- result
```

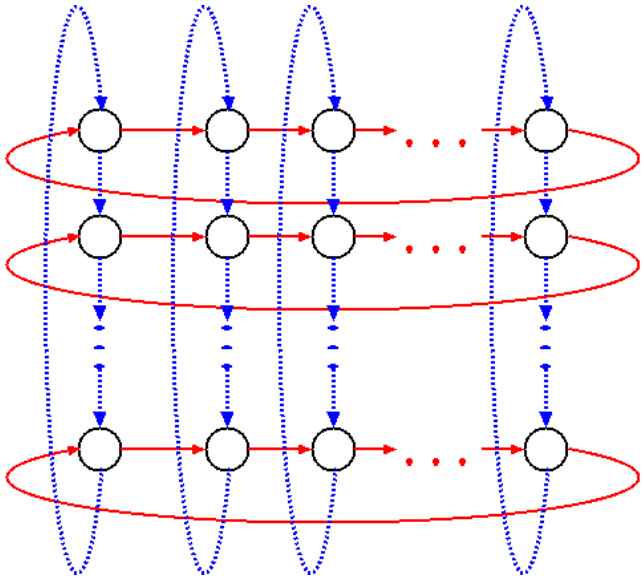


- Initialisation data `[[c]]`

- Ring-shaped neighbour communication in two dimensions

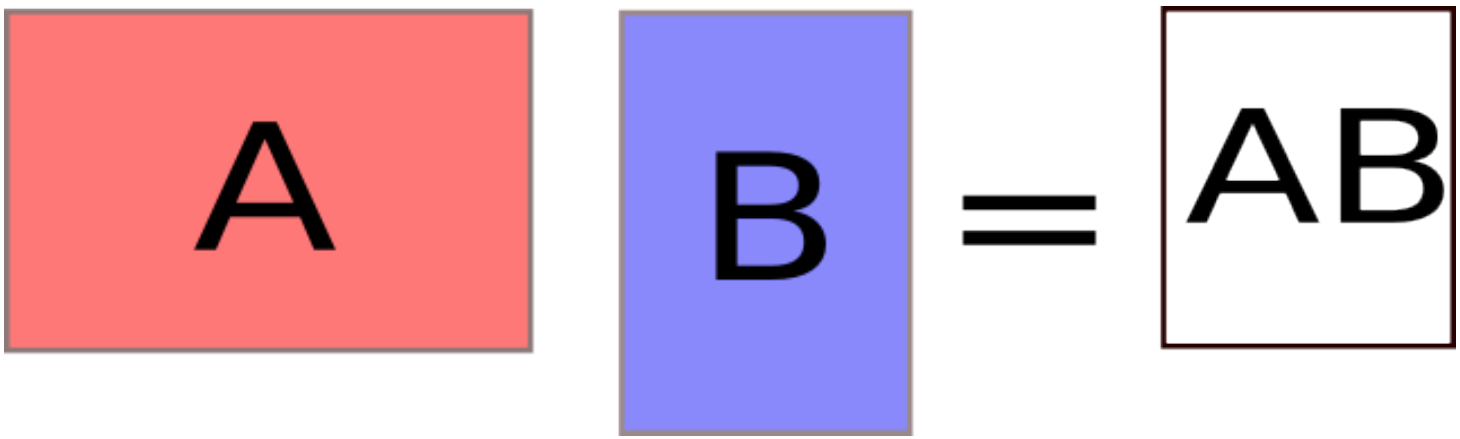
Process Topologies as Skeletons: Torus

```
torus ::
   -- node behaviour
  (c->[a]->[b] -> (d,[a],[b])) ->
   -- input (truncated to shortest)
  [[c]] -> [[d]] -- result
```



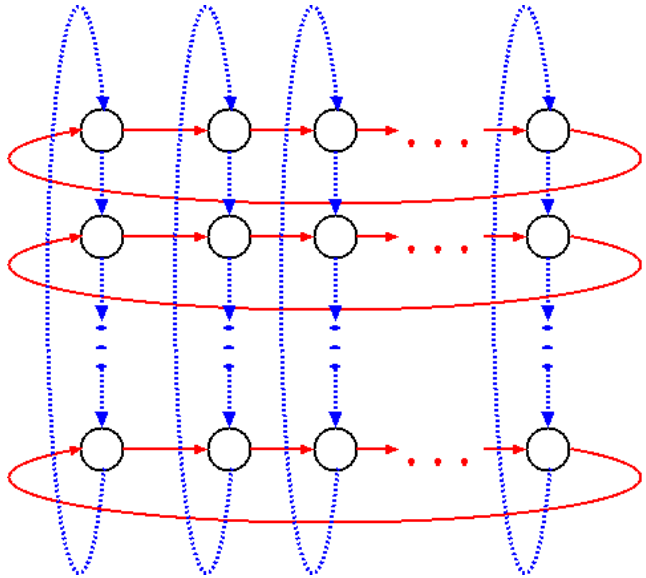
- Initialisation data `[[c]]`

- Ring-shaped neighbour communication in two dimensions
- Application: Matrix multiplication



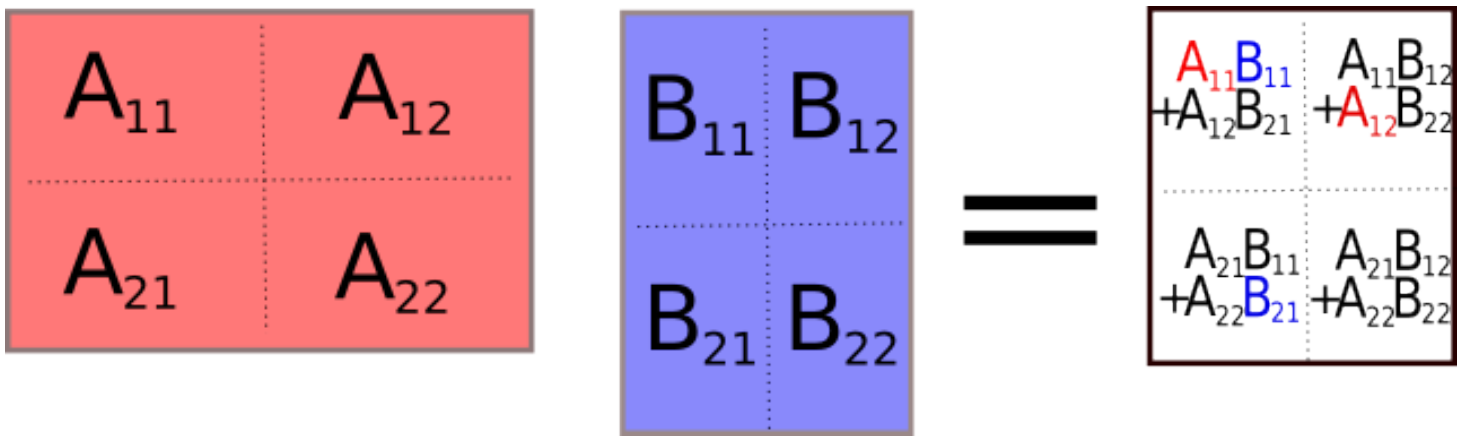
Process Topologies as Skeletons: Torus

```
torus ::
   -- node behaviour
  (c->[a]->[b] -> (d,[a],[b])) ->
   -- input (truncated to shortest)
  [[c]] -> [[d]] -- result
```



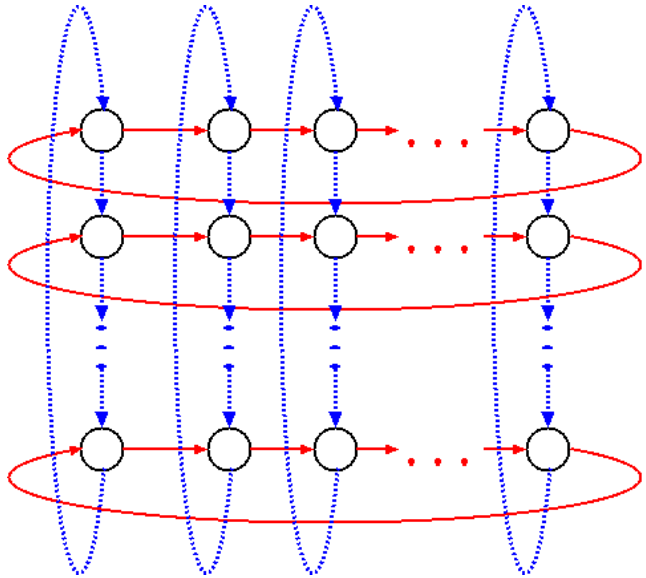
- Initialisation data `[[c]]`

- Ring-shaped neighbour communication in two dimensions
- Application: Matrix multiplication



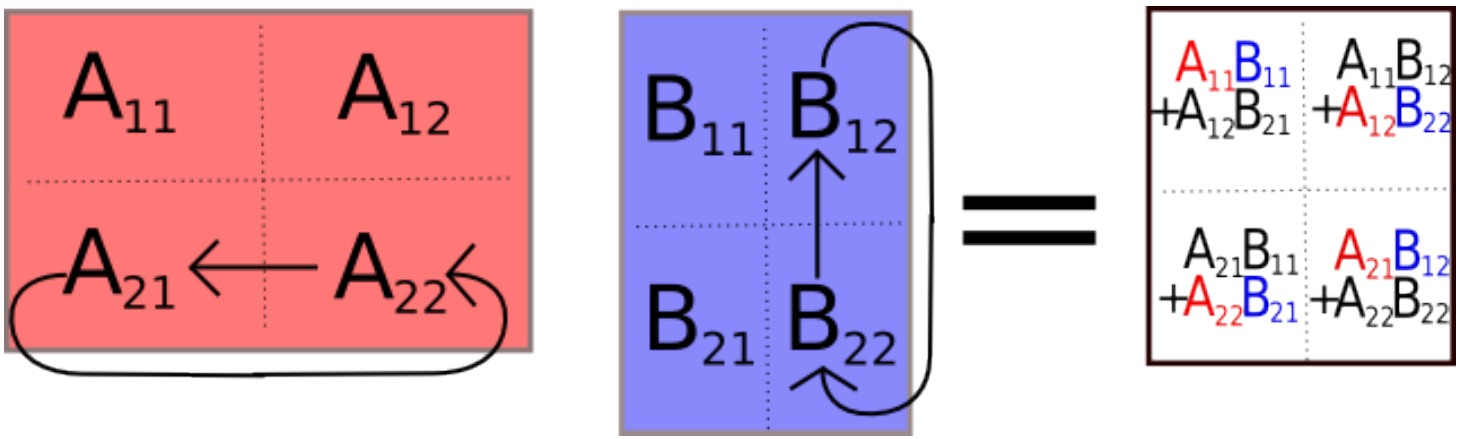
Process Topologies as Skeletons: Torus

```
torus ::
   -- node behaviour
  (c->[a]->[b] -> (d,[a],[b])) ->
   -- input (truncated to shortest)
  [[c]] -> [[d]] -- result
```



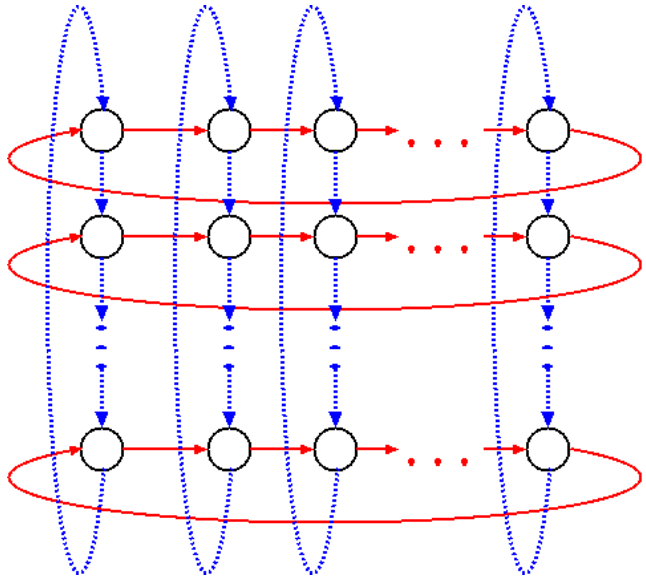
- Initialisation data `[[c]]`

- Ring-shaped neighbour communication in two dimensions
- Application: Matrix multiplication



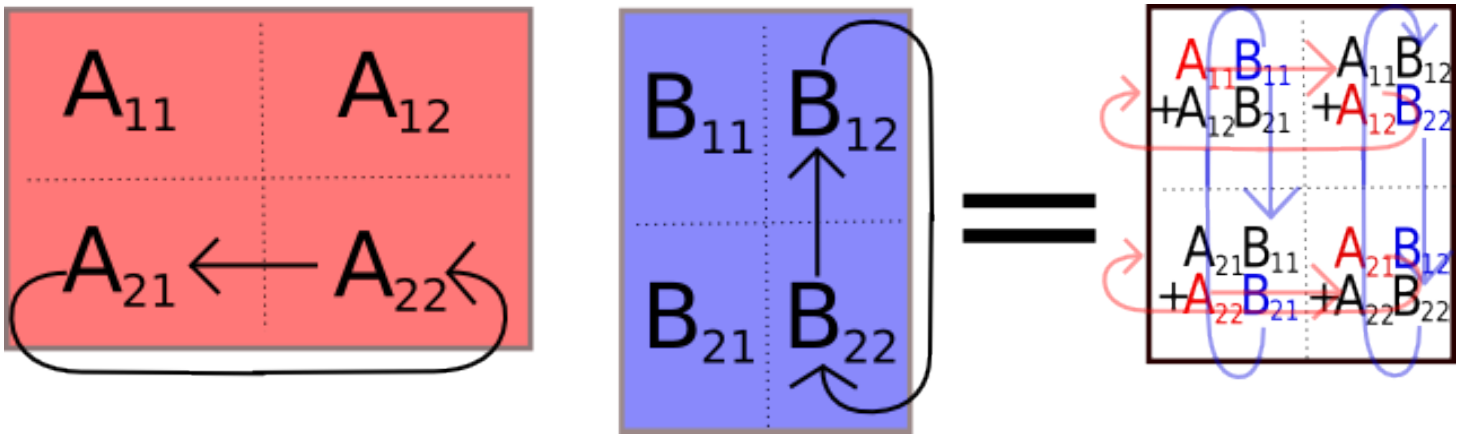
Process Topologies as Skeletons: Torus

```
torus ::
   -- node behaviour
  (c->[a]->[b] -> (d,[a],[b])) ->
   -- input (truncated to shortest)
  [[c]] -> [[d]] -- result
```



- Initialisation data `[[c]]`

- Ring-shaped neighbour communication in two dimensions
- Application: Matrix multiplication

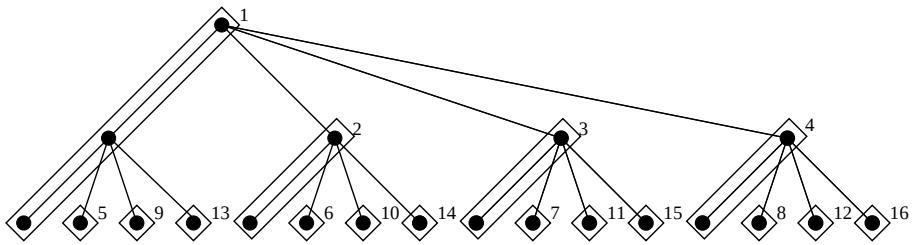


Outline

Two Algorithm-oriented Skeletons

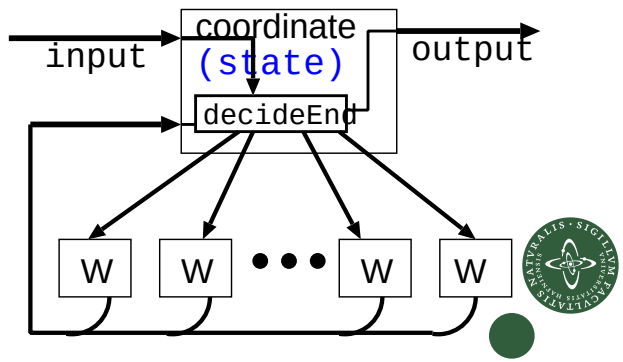
- Divide and conquer

```
divCon :: (a -> Bool) -> (a -> b)      -- trivial? / then solve
          -> (a -> [a]) -> ([b] -> b) -- split / combine
          -> a -> b                   -- input / result
```



- Iteration

```
iterateUntil :: (inp -> ([ws],[t],ms)) ->          -- split/init function
                (t -> State ws r) ->               -- worker function
                ([r] -> State ms (Either out [t])) -- manager function
                -> inp -> out
```



Divide and Conquer Skeletons

- General version: no assumptions on problem characteristics

```
divCon :: (a -> Bool) -> (a -> b)      -- trivial? / then solve
          -> (a -> [a]) -> ([b] -> b) -- split / combine
          -> a -> b                   -- input / result
divCon trivial solve split combine = ...
```

- Implementation will make (parallel?) recursive calls to itself (with same parameters as the initial call).

Divide and Conquer Skeletons

- General version: no assumptions on problem characteristics

```
divCon :: (a -> Bool) -> (a -> b)      -- trivial? / then solve
          -> (a -> [a]) -> ([b] -> b) -- split / combine
          -> a -> b                   -- input / result
divCon trivial solve split combine = ... -- you write one
```

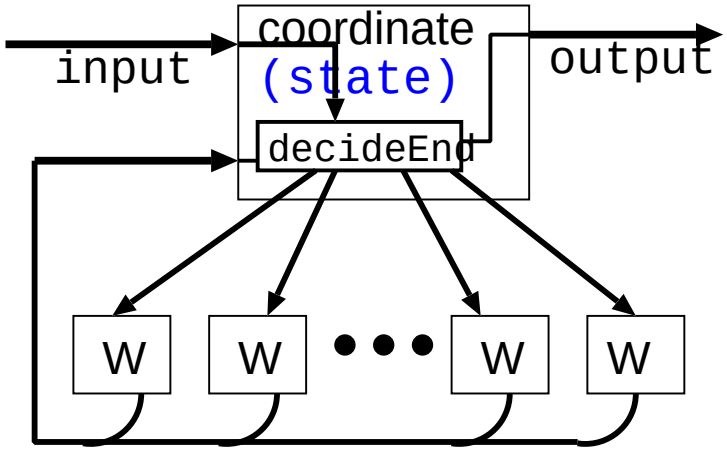
- Implementation will make (parallel?) recursive calls to itself (with same parameters as the initial call).

Exercise:

- Implement this general divide-and-conquer version.
  Write a sequential version first, then make recursive calls parallel.
  Add one `Int` parameter to limit the parallel depth.

Iteration Skeleton

- Fixed set of workers
- Lock-step execution, solving a set of tasks
- Manager decides end




```
iterateUntil :: (inp -> ([ws],[t],ms)) ->          -- split/init function
                (t -> State ws r) ->               -- worker function
                ([r] -> State ms (Either out [t])) -- manager function
                -> inp -> out
```

Worker: computes result `r` from task `t`

using and updating a local state `ws`

Manager: decides whether to continue,
based on master state `ms` and all worker results.

produce tasks for all workers

Outline

Summary

- Eden: Explicit parallel processes, mostly functional face
- Two levels of Eden: Skeleton implementation and skeleton use
  - Skeletons: High-level specification exposes parallel structure
  - and enables programmers to think in parallel patterns.
- Different skeleton categories (increasing abstraction)
  - Small-scale skeletons (map, fold, map-reduce, ...)
  - Process topology skeletons (ring, torus...)
  - Algorithmic skeletons (divide & conquer, iteration)

Summary

- Eden: Explicit parallel processes, mostly functional face
- Two levels of Eden: Skeleton implementation and skeleton use
  - Skeletons: High-level specification exposes parallel structure
  - and enables programmers to think in parallel patterns.
- Different skeleton categories (increasing abstraction)
  - Small-scale skeletons (map, fold, map-reduce, . . .)
  - Process topology skeletons (ring, torus. . .)
  - Algorithmic skeletons (divide & conquer, iteration)
- More information on Eden:

  `http://www.mathematik.uni-marburg.de/~eden`

  (`http://hackage.haskell.org/package/edenskel/`)

  (`http://hackage.haskell.org/package/edenmodules/`)