Skeleton-Based Parallel Programming

(and the language Eden)

Jost Berthold
berthold@diku.dk

Department of Computer Science
University of Copenhagen

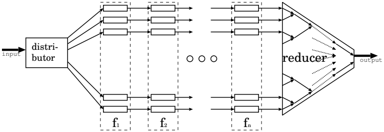
Chalmers University, March 29, 2012

High-level Parallel Programming

"The only thing that works for parallel programming is functional programming!"

Prof. Robert Harper, Carnegie Mellon University

`pipeline::[[a]->[a]] -> [a] -> [a]`

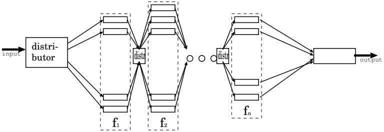


High-level Parallel Programming

"The only thing that works for parallel programming is functional programming!"

Prof. Robert Harper, Carnegie Mellon University

```
pipeline::[[a]->[a]] -> [a] -> [a]
```

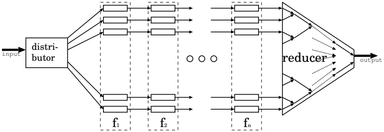


High-level Parallel Programming

"The only thing that works for parallel programming is functional programming!"

Prof. Robert Harper, Carnegie Mellon University

`pipeline::[[a]->[a]] -> [a] -> [a]`



Parallel + Functional = High-Level Parallel Programming

- . . . exposes algorithm structure and inherent parallelism,
- avoids typical problems of parallel programming,
- by abstraction over implementation details.
- High-level programming models:
  - Data parallel operations on container types (hidden parallelism)
  - Annotations on parallelisable expressions
  - Skeleton-based Programming describe algorithm / process structure as higher-order function(s)

About the Speaker: Jost Berthold

Research: Concepts/Implementation of Parallel Functional Programming

- Skeleton-based programming
- Parallelism Abstractions and Language Support
- Implementing parallel Haskell (Eden, GpH) since 2002

2003 Diploma (Computer Science) – Philipps-Universität Marburg
2008 Dr.rer.nat. (Computer Science) – Philipps-Universität Marburg
2008 Research Intern – Microsoft Research (GHC)
2008 PostDoc in SCIEnce – University of St.Andrews
2009 PostDoc in grid.dk – University of Copenhagen
2011 Researcher in HIPERFIT – University of Copenhagen

Eden: Explicit Process Control

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

Process abstraction: `process ::... (a -> b) -> Process a b`

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

Eden: Explicit Process Control

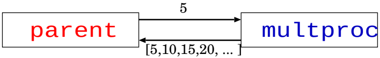
- Developed since 1996 in Marburg and Madrid
- Haskell, extended by communicating processes for coordination

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
```



or use: `($#) :: ... => (a -> b) -> a -> b`

Eden: Explicit Process Control

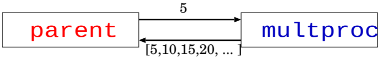
- Developed since 1996 in Marburg and Madrid
- Haskell, extended by communicating processes for coordination

Process abstraction: `process ::... (a -> b) -> Process a b`

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

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

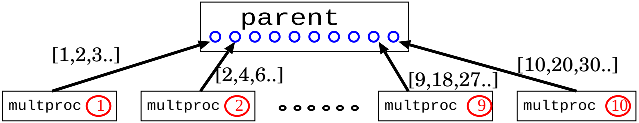
```
multiple5 = multproc # 5
```



or use: `($#) :: ... => (a -> b) -> a -> b`

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

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



or use: `(spawnF) :: ... => [a -> b] -> [a] -> b`

Eden: Explicit Process Control

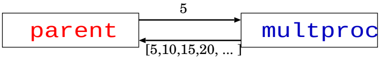
- Developed since 1996 in Marburg and Madrid
- Haskell, extended by communicating processes for coordination

Process abstraction: `process ::... (a -> b) -> Process a b`

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

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

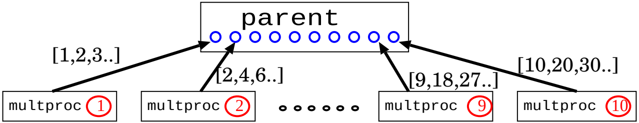
```
multiple5 = multproc # 5
```



or use: `($#) :: ... => (a -> b) -> a -> b`

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

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



or use: `(spawnF) :: ... => [a -> b] -> [a] -> b`

Full evaluation, stream communication, tuple concurrency.

A Small Eden Example

- 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.

A Small Eden Example

- 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.

A Small Eden Example

- 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.

Eden Constructs in a Nutshell

Eden main constructs: 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]
```

- Process instantiation (`( # )`) defines parent side
- Process abstraction (`process`) defines child side
- Helper function `spawn` to solve common demand problems.

More practical: combined abstraction/instantiation operator ($#)

```
( $# ) :: (Trans a, Trans b) => (a -> b) -> a -> b
spawnF :: (Trans a, Trans b) => [ a -> b ] -> [a] -> [b]
spawnF ps inputs = {- NOT REALLY -} zipWith ( $# )
```

Eden Constructs in a Nutshell

Eden main constructs: 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]
```

- Process instantiation (`( # )`) defines parent side
- Process abstraction (`process`) defines child side
- Helper function `spawn` to solve common demand problems.

More practical: combined abstraction/instantiation operator ($#)

```
( $# ) :: (Trans a, Trans b) => (a -> b) -> a -> b
spawnF :: (Trans a, Trans b) => [ a -> b ] -> [a] -> [b]
spawnF ps inputs = {- NOT REALLY -} zipWith ( $# )
```

Eden Constructs in a Nutshell

Eden main constructs: 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

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.

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.

Context: Parallel Languages extending 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

Context: Parallel Languages extending 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

  Similarities to the Par Monad‡ ("deterministic parallelism") (lower-level features, explicit communication)
- Concurrent Haskell‡, Eden implementation∗ (I/O monadic)
  explicit thread control and communication, full programmer control and responsibility

‡: shared memory, ∗: distributed memory

Overview

Motivation: Skeleton-Based Programming

You have already seen a nice example:

```
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 = ...
```

... even two versions!

```
divConq :: NFData sol =>
          (prob -> Bool)   -- indivisible?
          (prob -> [prob]) -- split into subproblems
          ([sol] -> sol)   -- join solutions
          (prob -> sol)    -- solve a subproblem
          (prob -> sol)
divConq indiv split combine solve input = ...
```

And another one, much simpler, much more common:

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

Motivation: Skeleton-Based Programming

You have already seen a nice example:

```
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 = ...
```

...even two versions!

```
divConq :: NFData sol =>
          (prob -> Bool)    -- indivisible?
          (prob -> [prob])  -- split into subproblems
          ([sol] -> sol)    -- join solutions
          (prob -> sol)     -- solve a subproblem
          (prob -> sol)
divConq indiv split combine solve input = ...
```

And another one, much simpler, much more common:

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

Motivation: Skeleton-Based Programming

You have already seen a nice example:

```
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 = ...
```

. . . even two versions!

```
divConq :: NFData sol =>
          (prob -> Bool)   -- indivisible?
          (prob -> [prob]) -- split into subproblems
          ([sol] -> sol)   -- join solutions
          (prob -> sol)    -- solve a subproblem
          (prob -> sol)
divConq indiv split combine solve input = ...
```

And another one, much simpler, much more common:

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

Motivation: Skeleton-Based Programming

You have already seen a nice example:

```
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 = ...
```

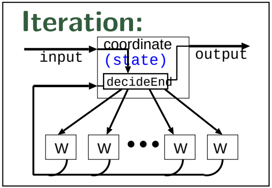
. . . even two versions!

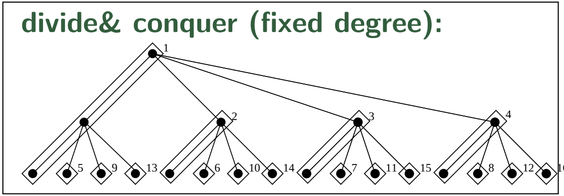
```
divConq :: NFData sol =>
          (prob -> Bool)    -- indivisible?
          (prob -> [prob]) -- split into subproblems
          ([sol] -> sol)    -- join solutions
          (prob -> sol)     -- solve a subproblem
          (prob -> sol)
divConq indiv split combine solve input = ...
```

And another one, much simpler, much more common:

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

Algorithmic Skeletons for Parallel Programming



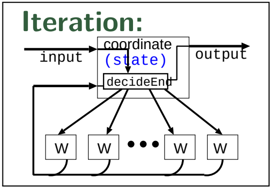


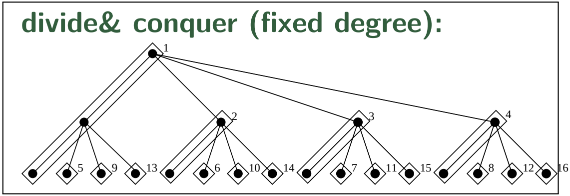
Boxes and lines – executable!

- Algorithmic Skeletons [Cole 1989]: abstract specification of. . .
- . . . algorithm structure as a higher-order function.
- Abstract over concrete tasks (embedded "worker" functions),
- hidden parallel optimised implementation(s) (machine-specific)

Different kinds of skeletons: small-scale, topological, algorithmic

Algorithmic Skeletons for Parallel Programming



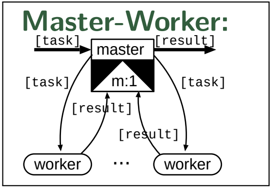


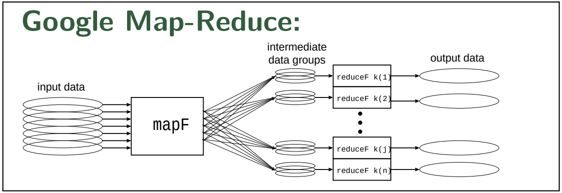
Boxes and lines – executable!

- Algorithmic Skeletons [Cole 1989]: abstract specification of. . .
- . . . algorithm structure as a higher-order function.
- Abstract over concrete tasks (embedded "worker" functions),
- hidden parallel optimised implementation(s) (machine-specific)

Different kinds of skeletons: small-scale, topological, algorithmic

Algorithmic Skeletons for Parallel Programming





Boxes and lines – executable!

- Algorithmic Skeletons [Cole 1989]: abstract specification of. . .
- . . . algorithm structure as a higher-order function.
- Abstract over concrete tasks (embedded “worker” functions),
- hidden parallel optimised implementation(s) (machine-specific)

Different kinds of skeletons: small-scale, topological, algorithmic

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

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
```

with commutative and associative operation.

- Parallel Scan:

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

reduction keeping the intermediate results.

- Parallel Map-Reduce:

combining transformation and groupwise reduction.

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
```

  with commutative and associative operation.

- Parallel 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
```

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    = ...
          distribute n = ... -- join . distribute n == id
```

Possible groupings: round-robin, in chunks

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    = ...
          distribute n = ... -- join . distribute n == id
```

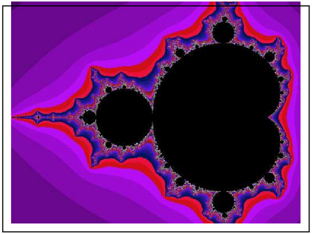
Possible groupings: round-robin, in chunks

An Example

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

Mandelbrot (Pseudocode)

```
mkPicture :: Int -> [[Word8]] -- binary pixels
mkPicture resolution = parMap computeRow (mkRows resolution)
```



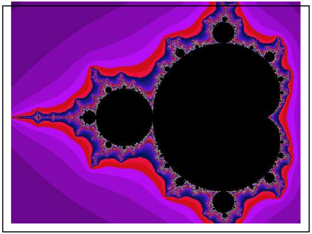
Simple chunking leads to load imbalance (task complexities differ)

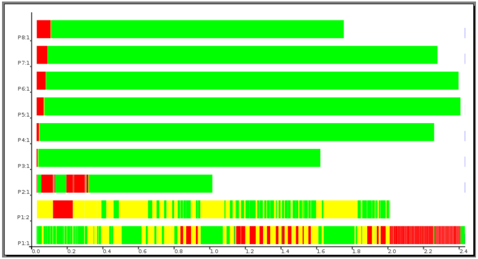
An Example

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

Mandelbrot (Pseudocode)

```
mkPicture :: Int -> [[Word8]] -- binary pixels
mkPicture resolution = parMap computeRow (mkRows resolution)
```





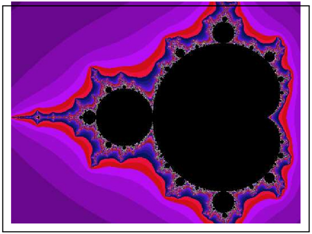
Simple chunking leads to load imbalance (task complexities differ)

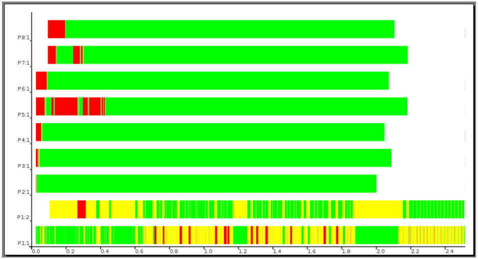
An Example

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

Mandelbrot (Pseudocode)

```
mkPicture :: Int -> [[Word8]] -- binary pixels
mkPicture resolution = parMap computeRow (mkRows resolution)
```





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

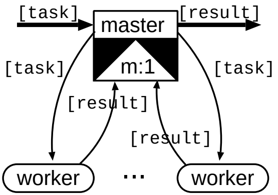
Master-Worker Skeleton

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 ⇒ more and more static task distribution
  Lower prefetch ⇒ dynamic load balance
- Result order needs to be reestablished!

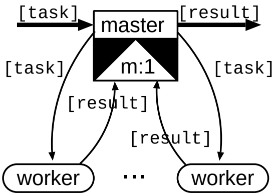
Master-Worker Skeleton

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 ⇒ more and more static task distribution
  Lower prefetch ⇒ dynamic load balance
- Result order needs to be reestablished!

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 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
```

- Associativity and neutral element (essential).
- commutativity (desired, more liberal distribution)
- need to narrow type of the reduce parameter function!
- . . . Alternative `fold` type: ...

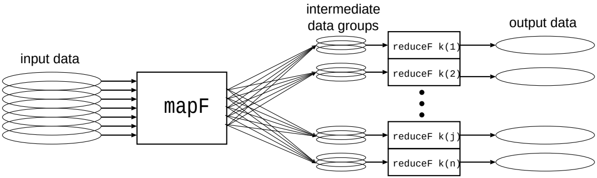
```
redF' :: [b] -> b
redF'    []    = neutral
redF' (x:xs) = foldl' redF x xs
```

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

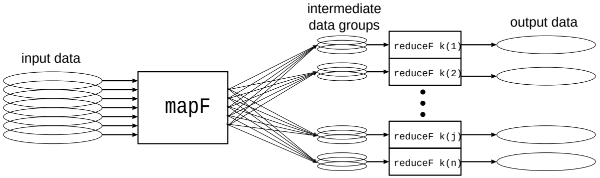
```
gMapRed :: (k1 -> v1 -> [(k2,v2)])      -- mapF
        -> (k2 -> [v2] -> Maybe v3) -- reduceF
        -> Map k1 v1 -> Map k2 v3   -- input / output
```



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
```

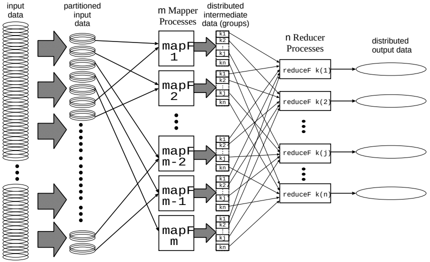


Document -> [(word,1)] -> word,count

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)
```

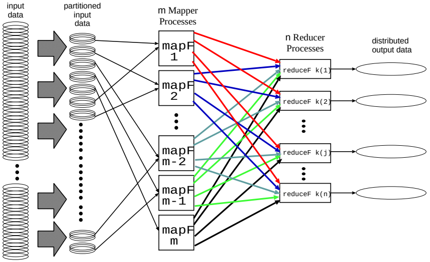
Google Map-Reduce (parallel)



R.Lämmel, Google's Map-Reduce Progr. 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
```

Google Map-Reduce (parallel)

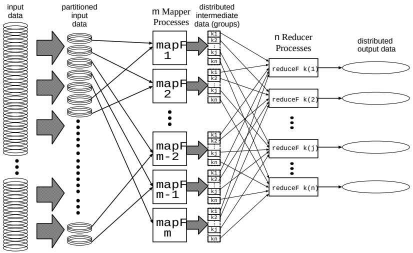


R.Lämmel, Google's Map-Reduce Progr. Model Revisited. In: SCP 2008

```
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
```

Google Map-Reduce (parallel): Properties

- reduceF associative and commutative
- Strictly speaking: **different types** in the reduction
- Keys k1: obsolete "bells and whistles"



Additional skeleton parameters (following Lämmel):

- Assignment of keys to reducers `k2 -> Int` (assumed $\in \{1..n\}$)
- Desired input size and estimation function `v1 -> Int`

R.Lämmel, Google's Map-Reduce Progr. 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
           -> [(k1,v1)] -> Map k2 v4   -- input / output
```

Example: Sum of Euler totient values

sumEuler

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

sumEuler with MapReduce

```
mapF key val    = [(0,phi val)]
reduceF _ list = Just (sum list)
```

Example: Sum of Euler totient values

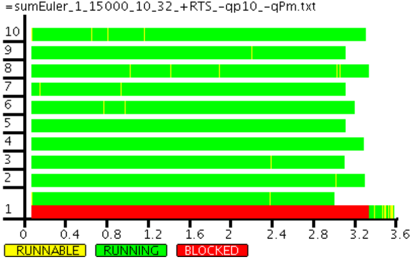
sumEuler

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

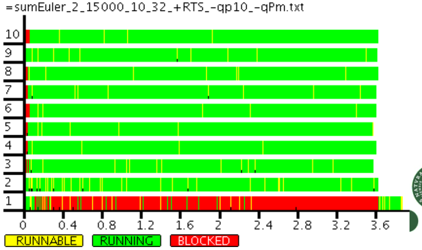
sumEuler with MapReduce

```
mapF key val    = [(0,phi val)]
reduceF _ list = Just (sum list)
```

Simple `map-fold` skeleton



Google Map-Reduce



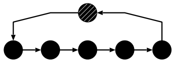
Overview

Process Topologies as Skeletons: Explicit Parallelism

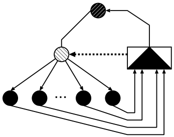
- describe typical patterns of parallel interaction structure
- (where node behaviour is the function argument)
- to structure parallel computations

Examples:

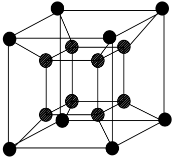
Pipeline/Ring:



Master/Worker:



Hypercube:

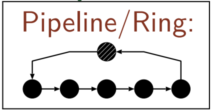


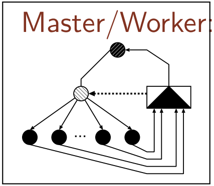
⇒ well-suited for functional languages (with explicit parallelism).
Skeletons can be implemented and applied in Eden.

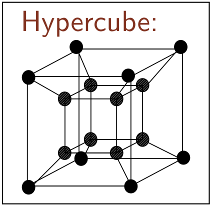
Process Topologies as Skeletons: Explicit Parallelism

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

Examples:

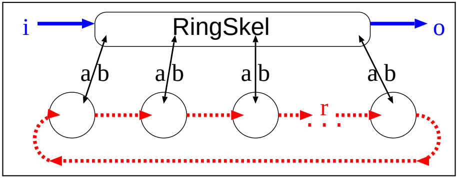






$\Rightarrow$ well-suited for functional languages (with explicit parallelism).
Skeletons can be implemented and applied in Eden.

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

Example: All Pairs Shortest Paths

(known as the Floyd-Warshall algorithm)

Adjacency Matrix Distance Matrix

$$\begin{pmatrix} 0 & w_{1,2} & w_{1,3} & \dots & w_{1,n} \\ w_{2,1} & 0 & w_{2,3} & \dots & w_{2,n} \\ w_{3,1} & w_{3,2} & 0 & \dots & w_{3,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{n,1} & w_{n,2} & w_{n,3} & \dots & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & d_{1,2} & d_{1,3} & \dots & d_{1,n} \\ d_{2,1} & 0 & d_{2,3} & \dots & d_{2,n} \\ d_{3,1} & d_{3,2} & 0 & \dots & d_{3,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d_{n,1} & d_{n,2} & d_{n,3} & \dots & 0 \end{pmatrix}$$

Floyd-Warshall: Update all rows k in parallel

```
ring_iterate :: Int -> Int -> Int -> [Int] -> [[Int]] -> ([Int],[[Int]])
ring_iterate size k i rowk rows
    | i > size =  (rowk, [])          -- finished
    | i == k   =  (result, rowk:rest) -- send own row
    | otherwise = (result, rowi:rest)
    where rowi:xs = rows
          (result, rest) = ring_iterate size k (i+1) nextrowk xs
          nextrowk | i == k    = rowk -- no update for own row
                   | otherwise = updaterow rowk rowi distki
          distki  = rowk!!(i-1)
```

Example: All Pairs Shortest Paths

(known as the Floyd-Warshall algorithm)

Adjacency Matrix → Distance Matrix

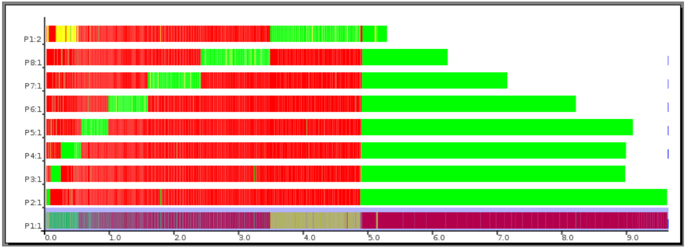
$$\begin{pmatrix} 0 & w_{1,2} & w_{1,3} & \dots & w_{1,n} \\ w_{2,1} & 0 & w_{2,3} & \dots & w_{2,n} \\ w_{3,1} & w_{3,2} & 0 & \dots & w_{3,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{n,1} & w_{n,2} & w_{n,3} & \dots & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & d_{1,2} & d_{1,3} & \dots & d_{1,n} \\ d_{2,1} & 0 & d_{2,3} & \dots & d_{2,n} \\ d_{3,1} & d_{3,2} & 0 & \dots & d_{3,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d_{n,1} & d_{n,2} & d_{n,3} & \dots & 0 \end{pmatrix}$$

Floyd-Warshall: Update all rows k in parallel

```
ring_iterate :: Int -> Int -> Int -> [Int] -> [[Int]] -> ([Int],[[Int]])
ring_iterate size k i rowk rows
    | i > size =  (rowk, [])           -- finished
    | i == k   =  (result, rowk:rest) -- send own row
    | otherwise = (result, rowi:rest)
    where rowi:xs = rows
          (result, rest) = ring_iterate size k (i+1) nextrowk xs
          nextrowk | i == k    = rowk -- no update for own row
                   | otherwise = updaterow rowk rowi distki
          distki  = rowk!!(i-1)
```

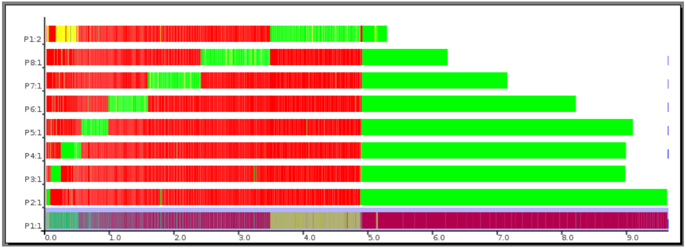
Trace of Warshall Program

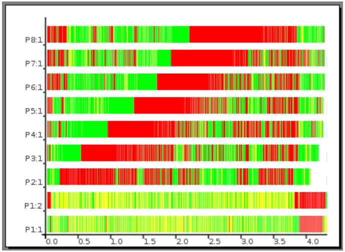
First version:



Trace of Warshall Program

First version:





with additional demand

Purely Functional Pipeline? (a topology skeleton)

Restricting to stages homogenous by their types

```
type Pipe a = [ [a] -> [a] ] -> [a] -> [a]
```

Can we program a pipeline with purely functional tools?

Tail-recursive:

```
pipeTR   []   xs = xs
pipeTR (f:fs) xs =
       pipeTR fs ( process f # xs)
```

Purely Functional Pipeline? (a topology skeleton)

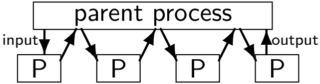
Restricting to stages homogenous by their types

```
type Pipe a = [ [a] -> [a] ] -> [a] -> [a]
```

Can we program a pipeline with purely functional tools?

Tail-recursive:

```
pipeTR   []   xs = xs
pipeTR (f:fs) xs =
       pipeTR fs ( process f # xs)
```



Purely Functional Pipeline? (a topology skeleton)

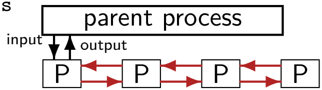
Restricting to stages homogenous by their types

```
type Pipe a = [ [a] -> [a] ] -> [a] -> [a]
```

Can we program a pipeline with purely functional tools?

Using inner recursion:

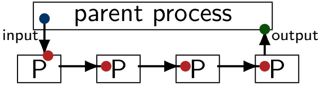
```
pipeR [] vals = vals
pipeR ps vals = (process (generatePipe ps)) # vals
generatePipe [p] vals  = p vals
generatePipe (p:ps) vals =
      (process (generatePipe ps)) # (p vals)
```



Pipeline (cont.d)

Recursion with dynamic reply channel:

```
ediRecPipe fs input
   = do (inCC,inC) <- createC
        (resC,res) <- createComm
        sendData (Instatiate 0) (doPipe inCC resC (reverse fs))
        fork (sendNFStream inC input)
        return res
doPipe incc resC (f:fs)
   = do (inC,input) <- createC
        if null fs then sendNF incc inC
             else sendData (Instantiate 0)
                     (doPipe incc inC fs)
        sendNFStream resC (f input)
```

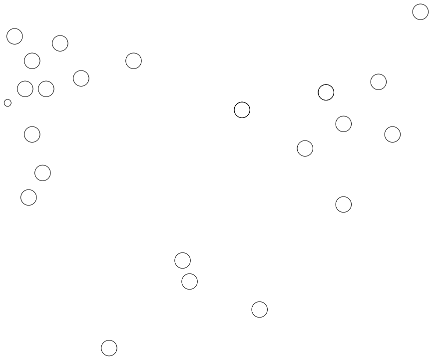


- Need to use explicit communication channels!
- Here written in EDI (IO-monadic Eden Implementation features)
- Can use Remote Data concept instead (not described here).

Overview

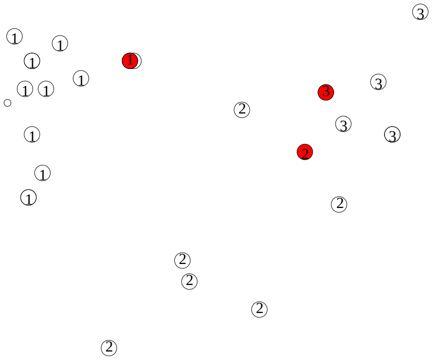
1. The Language Eden (in a nutshell)
2. Skeleton-Based Programming
   - The Skeleton Idea
   - Small-Scale Skeletons: Map and Reduce
   - Google Map-Reduce
3. Process Topologies as Skeletons
   - Process Topologies: Topology Skeletons
   - A Process Ring
   - Implementing a Pipeline...
4. More Skeletons
   - Google Map-Reduce revisited
   - Skeletons for Algorithmic Structure
5. Summary

K-Means Clustering



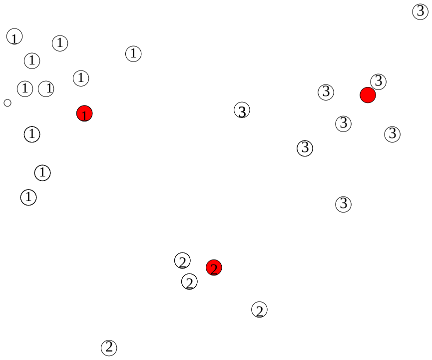
Given: Set of points. – Wanted: Centers of *n* clusters.

K-Means Clustering



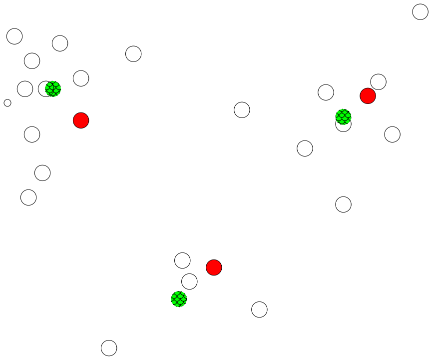
- randomly choose *n* centers
- assign pts to closest center

K-Means Clustering



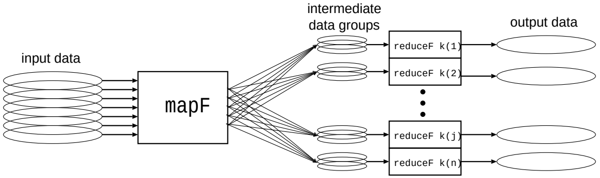
- randomly choose *n* centers
- assign pts to closest center
- compute centers of these groups
- iterate with new centers ...

K-Means Clustering



- Iterate this. . .
- . . . until centers do not change any more (finished)

K-Means Clustering using Google Map-Reduce



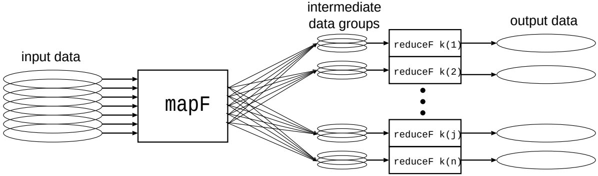
K-Means using Map-Reduce

```
kMeans :: Int -> [Vec] -> [Cent] -- no. of centers -> vectors -> centers
kMeans n vs = ...iteration of...
      newCenters = gMapReduce (mapKM oldCenters) reduceKM vs

mapKM ::  [Cent] -> Int -> Vec -> [(Int,Vec)]
mapKM cs _ vec = [(1+minIndex (map (distance vec) cs),vec)]

reduceKM :: Int -> [Vec] -> Maybe Cent
reduceKM _ vs = Just (center vs)
```

K-Means Clustering using Google Map-Reduce



K-Means using Map-Reduce

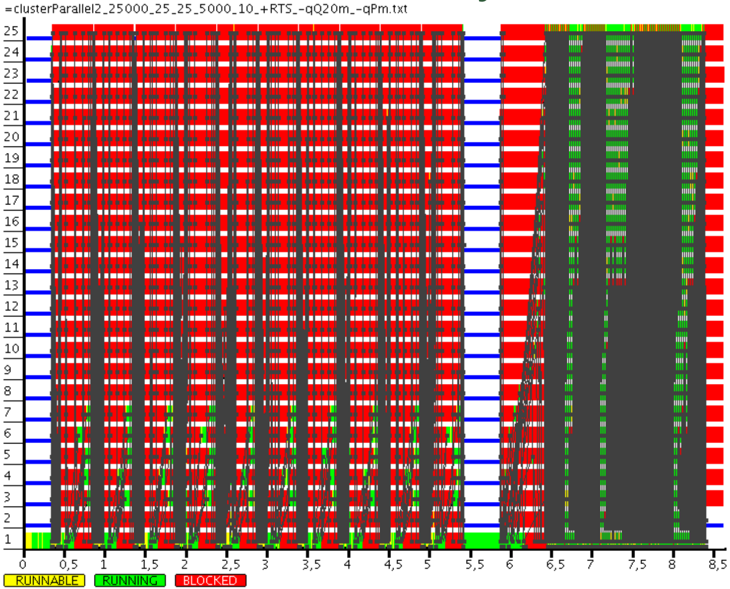
```
kMeans :: Int -> [Vec] -> [Cent] -- no. of centers -> vectors -> centers
kMeans n vs = ...iteration of...
       newCenters = gMapReduce (mapKM oldCenters) reduceKM vs

mapKM ::  [Cent] -> Int -> Vec -> [(Int,Vec)]
mapKM cs _ vec = [(1+minIndex (map (distance vec) cs),vec)]

reduceKM :: Int -> [Vec] -> Maybe Cent
reduceKM _ vs = Just (center vs)
```

But there are more clever ways. . .

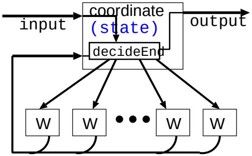
But there are more clever ways. . .



A Better K-Means Clustering: Iteration

Use an iteration skeleton!

Do not move the (unmodified!) huge data around all the time.



Iteration Skeleton

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

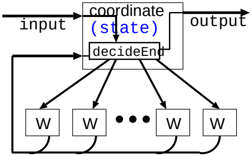
Worker: compute result r from task t
using and updating a local state

Manager: decide whether to continue,
based on master state and worker results.
produce tasks for all workers

A Better K-Means Clustering: Iteration

Use an iteration skeleton!

Do not move the (unmodified!) huge data around all the time.



Iteration Skeleton

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

Worker: compute result `r` from task `t` using and updating a local state

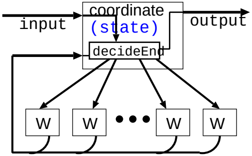
Manager: decide whether to continue, based on master state and worker results.

produce tasks for all workers

A Better K-Means Clustering: Iteration

Use an iteration skeleton!

Do not move the (unmodified!) huge data around all the time.



K-Means using iteration skeleton

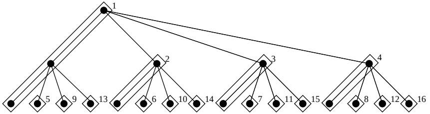
```
workerKMeans :: [Vec] -> [Cent] -> ([(Int,Cent)],[Vec])
workerKMeans vectors centroids = (newCs, vectors)
    where newCs = [ (length vs, if null vs then c else center vs)
                                | (c,vs) <- groups ]
          groups = groupByKey (assignVecs vectors centroids)

coordKMeans :: [Cent] -> [[(Int,Vec)]] -> Either [Cent] ([[Cent]],[Cent])
coordKMeans current workerOutputs
    | hardlyChanged current newCentroids = Left newCentroids
    | otherwise = Right (replicate nWorkers newCentroids,newCentroids)
  where newCentroids = map weightedAvg (transpose workerOutputs)
```

Other Algorithm-oriented Skeletons

- Iteration As just explained... (stateful worker and manager functions)
- Divide and conquer

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



- Backtracking (Tree search)

```
backtrack :: (a -> Maybe b)        -- maybe solve problem
             -> ( a  -> [a] )      -- refine problem one step
             -> a -> [b]           -- start problem / solutions
```

Backtracking: A Dynamically Growing Task Pool

- We use the master-worker skeleton with a small modification:

```
worker ::  task -> (Maybe result,[task])
```

- New tasks enqueued in dynamically growing task pool.
- Backtracking: Test decision alternatives until reaching a result.

Parallel SAT Solver

- Can a given logic formula be satisfied?
- Task pool starting with just one task (no variable assigned).

Backtracking: A Dynamically Growing Task Pool

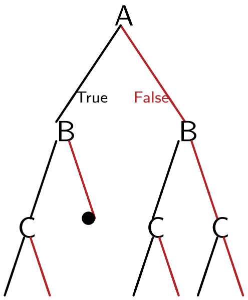
- We use the master-worker skeleton with a small modification:

```
worker :: task -> (Maybe result,[task])
```

- New tasks enqueued in dynamically growing task pool.
- Backtracking: Test decision alternatives until reaching a result.

Parallel SAT Solver

- Can a given logic formula be satisfied?
- Task pool starting with just one task (no variable assigned).



Backtracking: A Dynamically Growing Task Pool

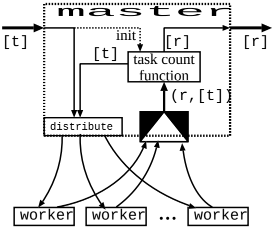
- We use the master-worker skeleton with a small modification:

```
worker ::  task -> (Maybe result,[task])
```

- New tasks enqueued in dynamically growing task pool.
- Backtracking: Test decision alternatives until reaching a result.

Parallel SAT Solver

- Can a given logic formula be satisfied?
- Task pool starting with just one task (no variable assigned).



- Stateful master with task counter:
  - consumes output of all workers
  - adds new tasks to task list
  - closes task list when counter reaches zero

Domain-Specific Skeletons: An Example

Orbit: Transitive closure under F:

Let M be a set, $F = \{f : M \rightarrow M\}$ a set of generator functions.
Compute for $S \subset M$: $\text{orbit}(S, F) = R \;\Leftrightarrow\; \forall_{r \in R}.\forall_{f \in F}.f(r) \in R$

Implementation aspects:

- Parallelise over generators or start set?
- How many elements, how many iterations expected?
- How large will the objects become?

```
orbit s fs = iterateUntil initWs doFs checkNew s
  where checkNew prev rs | all (`elem` prev) current = Left prev
                         | otherwise = (splitIntoN nw new, new)
          where current = foldl1 union rs
                new     = union prev current
        doFs () xs = ([f x | f <- fs, x <- xs],())
        initWs s nw = ..
```

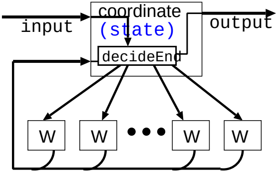
Domain-Specific Skeletons: An Example

Orbit: Transitive closure under F:

Let M be a set, $F = \{f : M \to M\}$ a set of generator functions.
Compute for $S \subset M$: $\text{orbit}(S, F) = R \;\Leftrightarrow\; \forall_{r \in R}.\forall_{f \in F}.f(r) \in R$

Implementation aspects:

- Parallelise over generators or start set?
- How many elements, how many iterations expected?
- How large will the objects become?




```
orbit s fs = iterateUntil initWs doFs checkNew s
  where checkNew prev rs | all (‘elem‘ prev) current = Left prev
                         | otherwise = (splitIntoN nw new, new)
            where current = foldl1 union rs
                  new     = union prev current
        doFs () xs = ([f x | f <- fs, x <- xs],())
        initWs s nw = ..
```

Overview

Summary

- Parallel + Functional = High-Level Parallel Programming
- Different skeleton categories (increasing abstraction)
  - Small-scale skeletons (map, fold, map-reduce, ...)
  - Process topology skeletons (pipeline, ring, ...)
  - Algorithmic skeletons (iteration, divide/conquer, backtracking)
- Parallel Skeletons enable programmers to think parallel
  - Clear view on functionality and parallel structure
  - High-level specification exposes parallel structure
- Implementation in parallel Haskell: easy integration, type safety

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

Summary

- Parallel + Functional = High-Level Parallel Programming
- Different skeleton categories (increasing abstraction)
  - Small-scale skeletons (map, fold, map-reduce, ...)
  - Process topology skeletons (pipeline, ring, ...)
  - Algorithmic skeletons (iteration, divide/conquer, backtracking)
- Parallel Skeletons enable programmers to think parallel
  - Clear view on functionality and parallel structure
  - High-level specification exposes parallel structure
- Implementation in parallel Haskell: easy integration, type safety
  More information: `http://www.mathematik.uni-marburg.de/~eden`