Eden: Parallel Processes, Patterns and Skeletons

Jost Berthold

berthold@diku.dk

Chalmers University of Technology, April 2014

Eden

Parallel
Functional
Programming

Contents

Contents

Learning Goals:

- Writing programs in the parallel Haskell dialect Eden;
- Reasoning about the behaviour of Eden programs;
- Applying the idea of skeleton-based programming;
- 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$^{\ddagger *}$ (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

$\ddagger$: 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, 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..]])
```

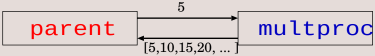
Essential Eden: Process Abstraction/Instantiation

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

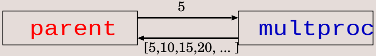
Essential Eden: Process Abstraction/Instantiation

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

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

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

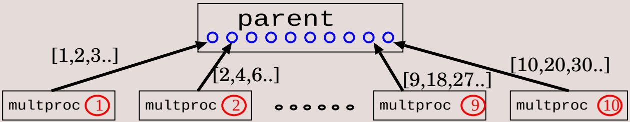
```
multiple5 = multproc # 5
```



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

Spawning many 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 $H = 5K$; where K is a Hamming smaller than H.

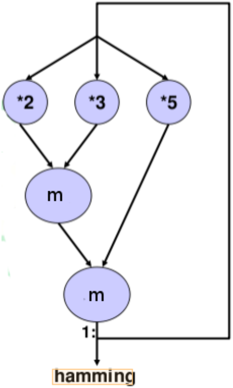
Basic Eden Exercise: Hamming Numbers

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 $H = 5K$; where K is a Hamming smaller than H.

- Write an Eden program that produces Hamming numbers using parallel processes. The program should take one argument n and produce the first n Hamming numbers.
- 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 in the homework:

- Divide and Conquer, as a higher-order function

```
divConq :: (prob -> Bool)         -- is the problem indivisible?
            -> (prob -> [prob])  -- split
            -> ([sol] -> sol)    -- join
            -> (prob -> sol)     -- solve a sub-problem
            -> (prob -> sol)
divConq indiv divide combine basecase = ...
```

  (this is just one version, more later. . .)

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

The Idea of Skeleton-Basked Parallelism

You have already seen one example in the homework:

- Divide and Conquer, as a higher-order function

```
divConq :: (prob -> Bool)          -- is the problem indivisible?
           -> (prob -> [prob])     -- split
           -> ([sol] -> sol)       -- join
           -> (prob -> sol)        -- solve a sub-problem
           -> (prob -> sol)
divConq indiv divide combine basecase = ...
```

(this is just one version, more later. . .)

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

And another one, much simpler, much more common:

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

The Idea of Skeleton-Basked Parallelism

You have already seen one example in the homework:

- Divide and Conquer, as a higher-order function

```
divConq :: (prob -> Bool)          -- is the problem indivisible?
           -> (prob -> [prob])   -- split
           -> ([sol] -> sol)     -- join
           -> (prob -> sol)      -- solve a sub-problem
           -> (prob -> sol)
divConq indiv divide combine basecase = ...
```

(this is just one version, more later. . .)

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

And another one, much simpler, much more common:

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

The Idea of Skeleton-Basked Parallelism

You have already seen one example in the homework:

- Divide and Conquer, as a higher-order function

```
divConq :: (prob -> Bool)         -- is the problem indivisible?
            -> (prob -> [prob])   -- split
            -> ([sol] -> sol)     -- join
            -> (prob -> sol)      -- solve a sub-problem
            -> (prob -> sol)
divConq indiv divide combine basecase = ...
```

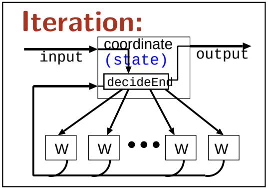
(this is just one version, more later. . .)

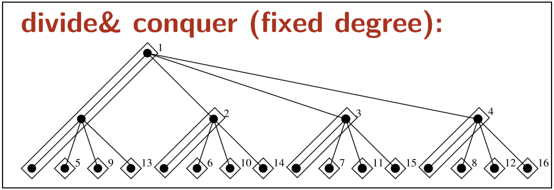
- Parallel structure (rose tree) exploited for parallelism
- Abstracted from concrete problem

And another one, much simpler, much more common:

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

Algorithmic Skeletons for Parallel Programming

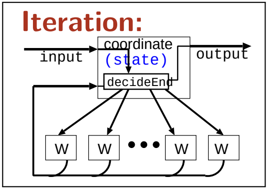


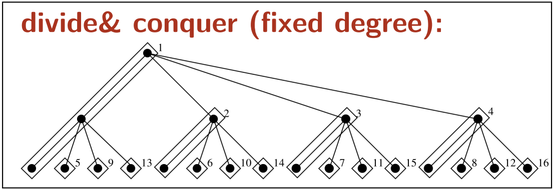


Algorithmic Skeletons [Cole 1989]: Boxes and lines – executable!

- Abstraction of algorithmic structure as a higher-order function

Algorithmic Skeletons for Parallel Programming

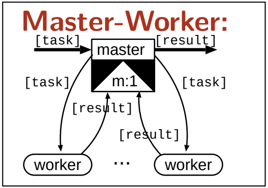


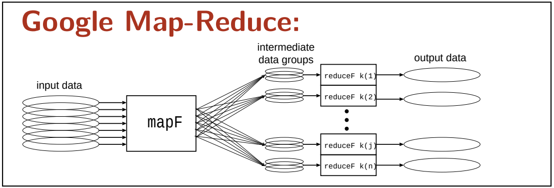


Algorithmic Skeletons [Cole 1989]: Boxes and lines – executable!

- 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

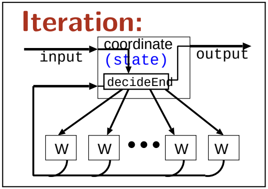


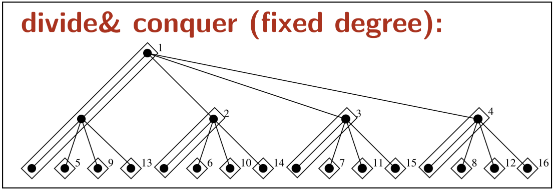


Algorithmic Skeletons [Cole 1989]: Boxes and lines – executable!

- 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 "embarrassingly 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 "embarrassingly 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 reduction.

Embarrassingly 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
```

Embarrassingly 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!

Embarrassingly 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

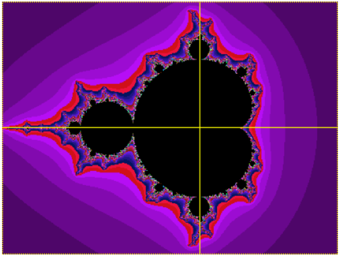
```
farm :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
farm f xs = join results
    where results = spawn (repeat (process (map f))) parts
          parts   = distrib noPe xs -- noPe, so use all nodes
          join    :: [[a]] -> [a]
          join     = ...
          distrib  :: Int -> [a] -> [[a]]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

Example application

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.. -- different options to distribute rows
```

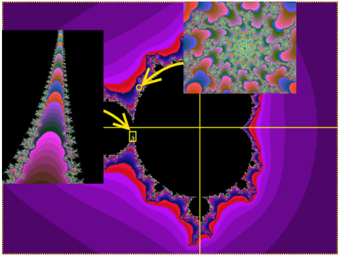


Example application

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.. -- different options to distribute rows
```



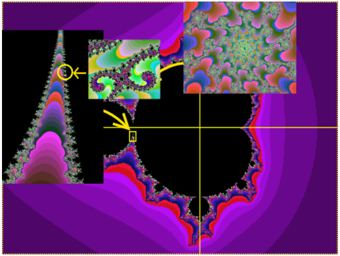
- Fractal properties (self-similarity)
- Colours indicate speed of divergence
    - Far out: diverges rapidly
    - Near 0: converges, or bounded

Example application

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.. -- different options to distribute rows
```



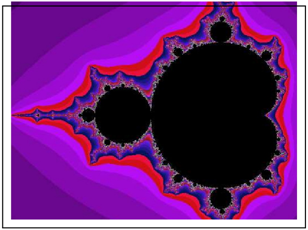
- Fractal properties (self-similarity)
- Colours indicate speed of divergence
  - Far out: diverges rapidly
  - Near 0: converges, or bounded
- Different rows expose different complexity

Example Application: 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..
```

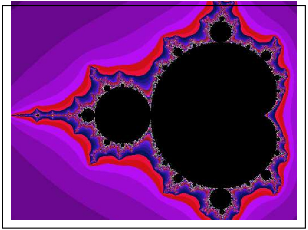


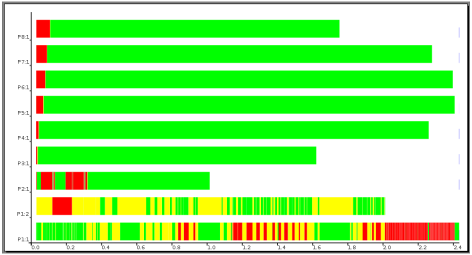
Example Application: Chunked Tasks

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..
```





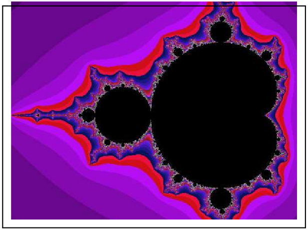
Simple chunking leads to load imbalance (task complexities differ)

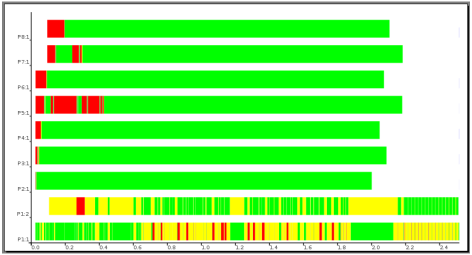
Example Application: Round-robin Tasks

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.

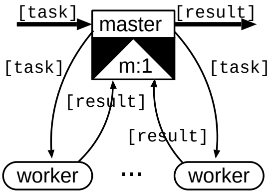
Dynamic load-balancing: Master-Worker Skeleton

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

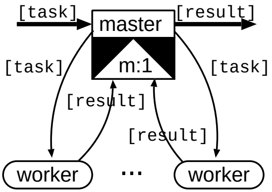
Dynamic load-balancing: Master-Worker Skeleton

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: 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 left or right, implying different reduction function.
  - To parallelise: break into sublists and pre-reduce in parallel.
    $\Rightarrow$ needs to drop direction and narrow type
  - 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 left or right, implying different reduction function.
- To parallelise: break into sublists and pre-reduce in parallel.
  $\Rightarrow$ needs to drop direction and narrow type
- 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

Map-Reduce skeleton

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

- need to narrow type of the reduce parameter function!
- Associativity and neutral element (essential).

Parallel Map-Reduce: Restrictions

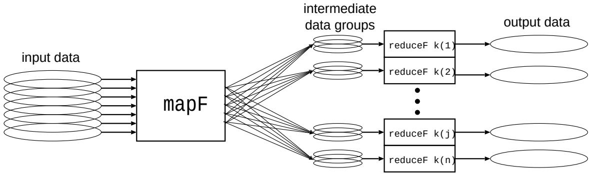
Map-Reduce skeleton

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

- need to narrow type of the reduce parameter function!
- Associativity and neutral element (essential).
- commutativity (desired, more liberal distribution)
  - `distrib` function may distribute in any order if `redF` commutative
    $\Rightarrow$ input consumed incrementally as a stream.
  - Otherwise: need to know list length ahead of time (inefficient).

Google Map-Reduce: Grouping Before Reduction

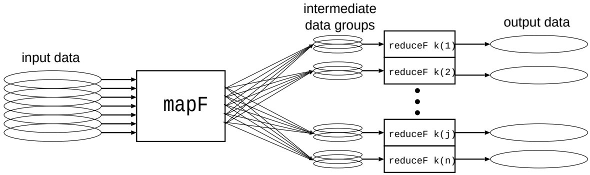
```
gMapRed :: (k1 -> v1 -> [(k2,v2)])      -- mapF
           -> (k2 -> [v2] -> Maybe v3)  -- reduceF
           -> Map k1 v1 -> Map k2 v3    -- input / output
```



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
```

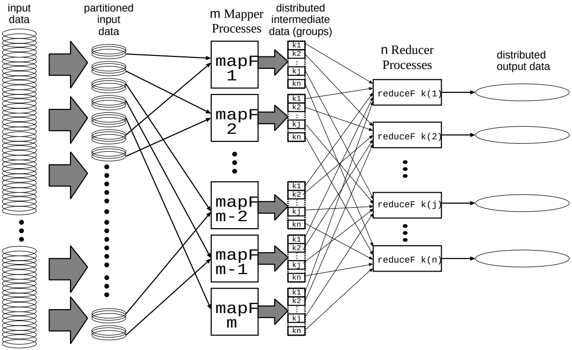



```
(uRL,document)    ==>    [(word,1)]  ==>    (word :-> count)
```

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
           -> Map 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 web pages, compute a dictionary to look up the pages that link to each page.

- URL Access Frequencies:
  Compute access counts for URLs from a set of web server log files.

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
           -> Map 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 web pages, compute a dictionary to look up the pages that link to each 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 access counts for URLs from a set of web server log files.

URL Access Frequency

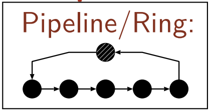
Input are all log entries, stating the requested URLs. As in word-occurrence: The map function emits (URL,1) pairs for requested URLs, the reduce functions sum the counts.

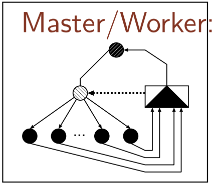
Outline

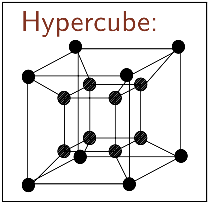
Process Topologies as Skeletons: Explicit Parallelism

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

Examples:





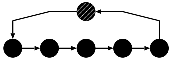


Process Topologies as Skeletons: Explicit Parallelism

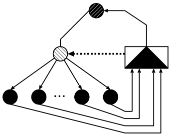
- describe typical patterns of parallel interaction structure
- (where node behaviour is the function argument)
- to structure parallel computations

Examples:

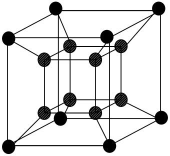
Pipeline/Ring:



Master/Worker:



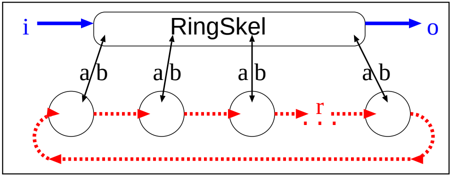
Hypercube:



$\Rightarrow$ well-suited for functional languages (with explicit parallelism).

Skeletons can be implemented and applied in Eden.

Process Topologies as Skeletons: Ring



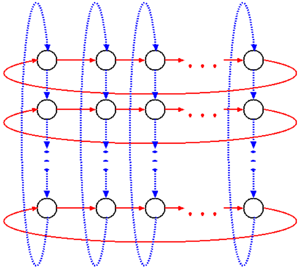

```
type RingSkel i o a b r = Int -> (Int -> i -> [a]) -> ([b] -> o) ->
                           ((a,[r]) -> (b,[r])) -> i -> o

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

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

Process Topologies as Skeletons: Torus

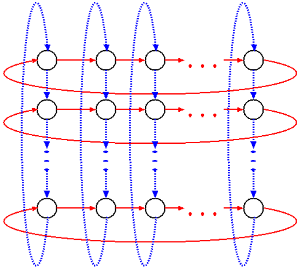
```
torus ::
   -- node behaviour
  (c->[a]->[b] -> (d,[a],[b])) ->
   -- input (truncated to shortest)
  [[c]] -> [[d]] -- result
```



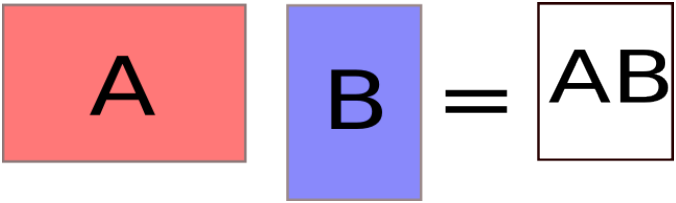
- Initialisation data `[[c]]`
- Ring-shaped neighbour communication in two dimensions

Process Topologies as Skeletons: Torus

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

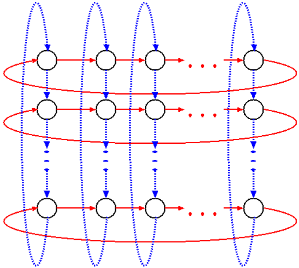


- Initialisation data `[[c]]`
- Ring-shaped neighbour communication in two dimensions
- Application: Matrix multiplication

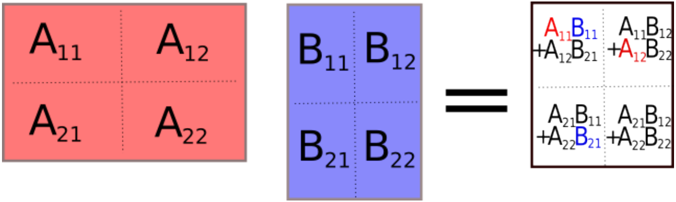


Process Topologies as Skeletons: Torus

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

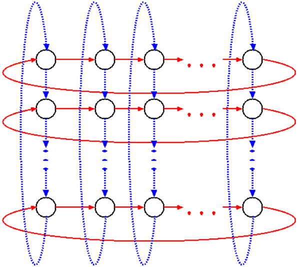


- Initialisation data `[[c]]`
- Ring-shaped neighbour communication in two dimensions
- Application: Matrix multiplication

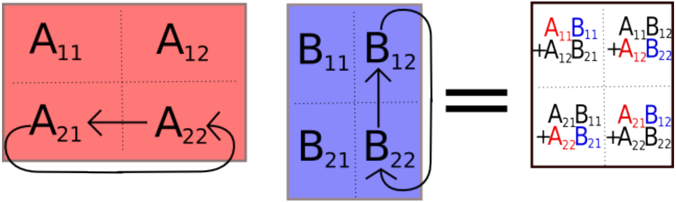


Process Topologies as Skeletons: Torus

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

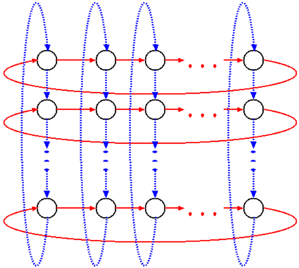


- Initialisation data `[[c]]`
- Ring-shaped neighbour communication in two dimensions
- Application: Matrix multiplication

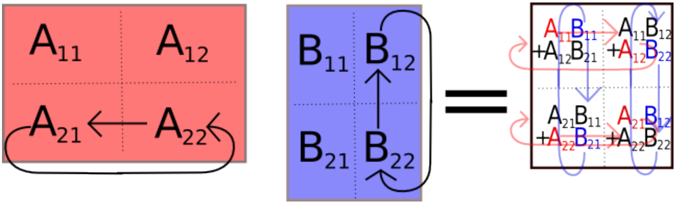


Process Topologies as Skeletons: Torus

```
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

Algorithm-oriented Skeletons

Divide and conquer

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

Iteration

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

Backtracking (Tree search)

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

Divide and Conquer Skeletons

- Mary, slide 66 in strategies lecture: binary divide&conquer

```
divConq indiv split join f prob = undefined
divCon :: (a -> b) -> a      -- base case fct., input
            -> (a -> Bool)     -- parallel threshold
            -> (b -> b -> b)  -- combine
            -> (a -> Maybe (a,a)) -- divide
            -> b
```

- Simon Marlow: slide 53, with a more general version

```
divConq :: (prob -> Bool)        -- is the problem indivisible?
            -> (prob -> [prob])  -- split
            -> ([sol] -> sol)    -- join
            -> (prob -> sol)     -- solve a sub-problem
            -> (prob -> sol)
```

Divide and Conquer Skeletons

- Mary, slide 66 in strategies lecture: binary divide&conquer

```
divConq indiv split join f prob = undefined
divCon :: (a -> b) -> a      -- base case fct., input
           -> (a -> Bool)     -- parallel threshold
           -> (b -> b -> b)   -- combine
           -> (a -> Maybe (a,a)) -- divide
           -> b
```

- Simon Marlow: slide 53, with a more general version

```
divConq :: (prob -> Bool)          -- is the problem indivisible?
           -> (prob -> [prob])  -- split
           -> ([sol] -> sol)    -- join
           -> (prob -> sol)     -- solve a sub-problem
           -> (prob -> sol)
```

...so here is my version:

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

Simple Divide & Conquer

Divide & Conquer Skeleton (simple general version)

```
dc_c depth trivial solve split combine x
   = if depth < 1 then seqDC x
       else if trivial x then solve x
              else childRs `seq` -- early demand on children results
                            combine x (myR : childRs)
    where myself = dc_c (depth - 1) trivial solve split combine
          seqDC x     = if trivial x then solve x
                          else combine x (map seqDC (split x))
          (mine:rest) = split x
          myR = myself mine
          childRs = parMapEden myself rest
```

Simple Divide & Conquer

Divide & Conquer Skeleton (simple general version)

```
dc_c depth trivial solve split combine x
   = if depth < 1 then seqDC x
       else if trivial x then solve x
              else childRs `seq` -- early demand on children results
                            combine x (myR : childRs)
  where myself = dc_c (depth - 1) trivial solve split combine
        seqDC x     = if trivial x then solve x
                          else combine x (map seqDC (split x))
        (mine:rest) = split x
        myR = myself mine
        childRs = parMapEden myself rest
```

Room for optimisation:

- Number of sub-problems often fixed by the algorithm
- Processes should be placed evenly on all machines

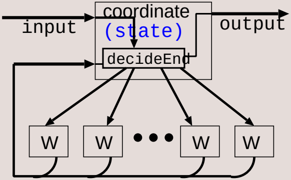
The Eden skeleton library contains many variants.

http://hackage.haskell.org/package/edenskel/

Parallel iteration (an algorithmic skeleton)

Iterated parallel `map` on tasks

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



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

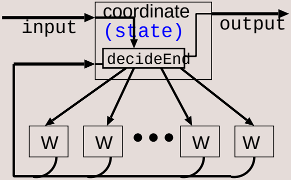
Manager: decide whether to continue,
based on master state `ms` and worker results `[r]`.

produce tasks `[t]` for all workers

Parallel iteration (an algorithmic skeleton)

Iterated parallel `map` on tasks

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



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

Manager: decide whether to continue, based on master state `ms` and worker results `[r]`.

produce tasks `[t]` for all workers

Applications: N-body, K-means clustering, genetic algorithms. . .

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.

Backtracking: A Dynamically Growing Task Pool

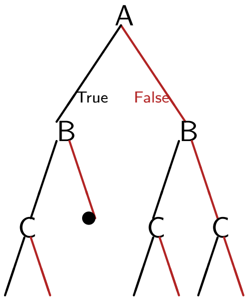
- We use the master-worker skeleton with a small modification:

```
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

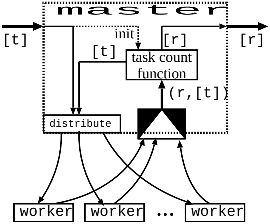
- We use the master-worker skeleton with a small modification:

```
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
  - add new tasks to task list
  - shutdown when counter reaches zero

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/
http://hackage.haskell.org/package/edentv/

add-on material ahead

Example: All Pairs Shortest Paths (Floyd-Warshall)

Adjacency Matrix

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

Distance Matrix

Example: All Pairs Shortest Paths (Floyd-Warshall)

Adjacency Matrix

Distance Matrix

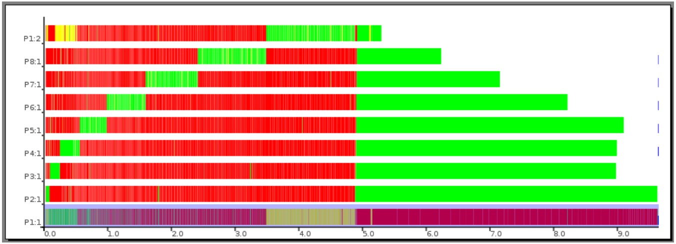
$$\begin{pmatrix} 0 & w_{1,2} & w_{1,3} & \dots & w_{1,n} \\ w_{2,1} & 0 & w_{2,3} & \dots & w_{2,n} \\ w_{3,1} & w_{3,2} & 0 & \dots & w_{3,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{n,1} & w_{n,2} & w_{n,3} & \dots & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & d_{1,2} & d_{1,3} & \dots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

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)
```

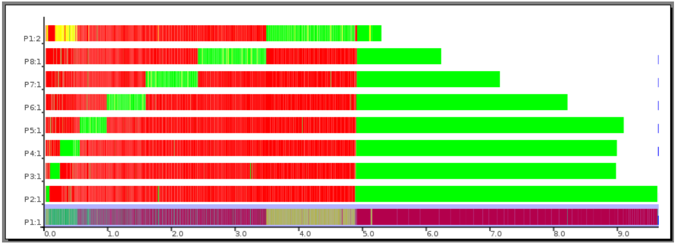
Trace of Warshall Program

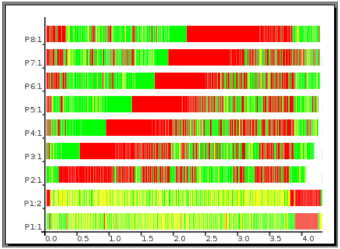
First version:



Trace of Warshall Program

First version:



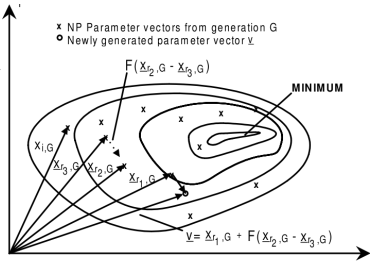


with additional demand

Differential Evolution [Price/Storn] with iteration skeleton

Worker: Mutate and select

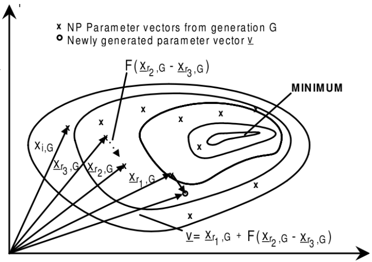
- Randomly choose C as either the best known (50%) or a random candidate
- Add weighted difference of 2 other candidates: $C'_i = C + \gamma(C_{r_1} - C_{r_2})$
- Using fitness function f: retain C_i if better (minimising: $f(C_i) < f(C'_i)$).
- State: random gen., local candidates



Differential Evolution [Price/Storn] with iteration skeleton

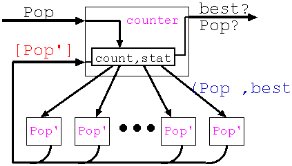
Worker: Mutate and select

- Randomly choose C as either the best known (50%) or a random candidate
- Add weighted difference of 2 other candidates: $C_i' = C + \gamma(C_{r_1} - C_{r_2})$
- Using fitness function f: retain C_i if better (minimising: $f(C_i) < f(C_i')$).
- State: random gen., local candidates



Manager: Collect/redistribute, identify best

- Termination:
  - when best/all cand. good enough,
  - or after n iteration steps
- State: Iteration counter



Eden usage example

Compile example, (with tracing -eventlog):

```
berthold@bwlf01$ COMPILER  -parcp -eventlog -O2 -rtsopts --make mandel.hs
[1 of 2] Compiling ParMap           ( ParMap.hs, ParMap.o )
[2 of 2] Compiling Main             ( mandel.hs, mandel.o )
Linking mandel ...
```

Run, second run with tracing:

```
berthold@bwlf01$  ./mandel 0 200 1 -out +RTS -qp4  > out.ppm
==== Starting parallel execution on 4 processors ...
berthold@bwlf01$ ./mandel 0 50 1 +RTS -qp4 -l
==== Starting parallel execution on 4 processors ...
Done (no output)
Trace post-processing...
  adding: berthold=mandel#1.eventlog (deflated 65%)
  adding: berthold=mandel#2.eventlog (deflated 59%)
  adding: berthold=mandel#3.eventlog (deflated 58%)
  adding: berthold=mandel#4.eventlog (deflated 58%)
berthold@bwlf01$ edentv berthold\=mandel_0_50_1_+RTS_-qp4_-l.parevents
```