Eden: Parallel Processes, Patterns and Skeletons

Jost Berthold
berthold@diku.dk
Department of Computer Science
Contents

1 The Language Eden (in a nutshell)
2 Skeleton-Based Programming
3 Small-Scale Skeletons: Map and Reduce
4 Process Topologies as Skeletons
5 Algorithm-Oriented Skeletons: Two Classics
6 Summary
Contents

1. The Language Eden (in a nutshell)
2. Skeleton-Based Programming
3. Small-Scale Skeletons: Map and Reduce
4. Process Topologies as Skeletons
5. Algorithm-Oriented Skeletons: Two Classics
6. Summary

Learning Goals:

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

```haskell
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)

```haskell
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
```
**Process Abstraction:** \[ \text{process} :: \ldots \ (a \to b) \to \text{Process} \ a \ b \]

\[
multproc = \text{process} (\\lambda x \to [x \times k \mid k \leftarrow [1, 2..]])
\]
Essential Eden: Process Abstraction/Instantiation

**Process Abstraction:** \[ \text{process} :: (a \to b) \to \text{Process} \ a \ b \]

\[ \text{multproc} = \text{process} \ (\lambda x \to [x\times k \mid k \leftarrow [1,2..]]) \]

**Process Instantiation:** \[ (\#) :: \text{Process} \ a \ b \to a \to b \]

\[ \text{multiple5} = \text{multproc} \# 5 \]

- Full evaluation of argument (concurrent) and result (parallel)
- Stream communication for lists
Essential Eden: Process Abstraction/Instantiation

Process Abstraction: \( \text{process} :: \ldots (a \rightarrow b) \rightarrow \text{Process} \ a \ b \)

\[
\text{multproc} = \text{process} (\lambda x \rightarrow [x \times k \mid k \leftarrow [1,2..]])
\]

Process Instantiation: \( \# :: \ldots \text{Process} \ a \ b \rightarrow a \rightarrow b \)

\[
\text{multiple5} = \text{multproc} \ # \ 5
\]

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

Spawning multiple processes: \( \text{spawn} :: \ldots \left[\text{Process} \ a \ b\right] \rightarrow \left[a\right] \rightarrow \left[b\right] \)

\[
\text{multiples} = \text{spawn} \ (\text{replicate} \ 10 \ \text{multproc}) \ [1..10]
\]
A Small Eden Example\textsuperscript{1}

- Subexpressions evaluated in parallel
- ...in different \texttt{processes} with separate heaps

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

\texttt{simpleeden.hs}

\begin{verbatim}
main = do args <- getArgs
  let [first_stuff,other_stuff] = spawnF [f_expensive, g_expensive] args
  putStrLn (show first_stuff ++ '
':show other_stuff)
\end{verbatim}

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

\textsuperscript{1}(compiled with option -parcp or -parmpi)
A Small Eden Example

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

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

...which will not produce any speedup!
```

\[simpleeden.hs\]

1(compiled with option -parcp or -parmpi)
A Small Eden Example

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

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

...which will not produce any speedup!

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

\[ \{2^i \cdot 3^j \cdot 5^k \mid i, j, k \in \mathbb{N}\} \]
Basic Eden Exercise: Hamming Numbers

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

\[
\{ 2^i \cdot 3^j \cdot 5^k \mid i, j, k \in \mathbb{N} \}
\]

Dijkstra:
The first Hamming number is 1. Each following Hamming number \( H \) can be written as \( H = 2K, H = 3K, \) or \( H = 5K; \) with a suitable smaller Hamming number \( K. \)
Basic Eden Exercise: Hamming Numbers

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

\[ \{ 2^i \cdot 3^j \cdot 5^k \mid i, j, k \in \mathbb{N} \} \]

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

- Write an Eden program that produces Hamming numbers using parallel processes. The program should take one argument \( n \) and produce the numbers up to position \( n \).
- Observe the parallel behaviour of your program using EdenTV.
Non-Functional Eden Constructs for Optimisation

Location-Awareness: \( \text{noPe}, \text{selfPe} :: \text{Int} \)

\[
\text{spawnAt} :: (\text{Trans } a, \text{Trans } b) \Rightarrow [\text{Int}] \rightarrow [\text{Process } a \ b] \rightarrow [a] \rightarrow [b]
\]

\[
\text{instantiateAt} :: (\text{Trans } a, \text{Trans } b) \Rightarrow \text{Int} \rightarrow \text{Process } a \ b \rightarrow a \rightarrow \text{IO } b
\]
Non-Functional Eden Constructs for Optimisation

Location-Awareness: \texttt{noPe}, \texttt{selfPe} :: \texttt{Int}

\texttt{spawnAt} :: (\texttt{Trans} \texttt{a}, \texttt{Trans} \texttt{b}) \to [\texttt{Int}] \to [\texttt{Process} \texttt{a} \texttt{b}] \to [\texttt{a}] \to [\texttt{b}]
\texttt{instantiateAt} :: (\texttt{Trans} \texttt{a}, \texttt{Trans} \texttt{b}) \to\texttt{Int} \to \texttt{Process} \texttt{a} \texttt{b} \to \texttt{a} \to \texttt{IO} \texttt{b}

Explicit communication using primitive operations (monadic)

\texttt{data ChanName = Comm (Channel \texttt{a} \to \texttt{a} \to \texttt{IO} () )}
\texttt{createC :: IO (Channel \texttt{a} , \texttt{a})}

\texttt{class NFData \texttt{a} => \texttt{Trans} \texttt{a} where}
\texttt{write :: \texttt{a} \to \texttt{IO} ()}
\texttt{createComm :: IO (ChanName \texttt{a}, \texttt{a})}
\texttt{createComm = do (cx,x) <- createC}
\texttt{return (Comm (sendVia cx) , x)}

Nondeterminism! \texttt{merge} :: [[\texttt{a}]] \to [\texttt{a}]

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

1 The Language Eden (in a nutshell)

2 Skeleton-Based Programming

3 Small-Scale Skeletons: Map and Reduce

4 Process Topologies as Skeletons

5 Algorithm-Oriented Skeletons: Two Classics

6 Summary
The Idea of Skeleton-Basked Parallelism

You have already seen one example:

- **Divide and Conquer, as a higher-order function**
  ```haskell
  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 = ...
  ```
  (type will be modified later)
- **Parallel structure (binary tree) exploited for parallelism**
- **Abstracted from concrete problem**
The Idea of Skeleton-Basked Parallelism

You have already seen one example:

- **Divide and Conquer, as a higher-order function**

  ```haskell
  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 = ...
  ```

  (type will be modified later)

- **Parallel structure (binary tree) exploited for parallelism**

- **Abstracted from concrete problem**

And another one, much simpler, much more common:

```haskell
parMap :: (a->b) -> [a] -> [b]
```
Algorithmic Skeletons for Parallel Programming

Iteration:

- Input
- Coordinate (state)
- Decide
- Output

divide & conquer (fixed degree):

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

- Abstraction of algorithmic structure as a higher-order function
Algorithmic Skeletons for Parallel Programming

**Iteration:**

- **input** → decideEnd
- **coordinate** → output
- **W** → **W** → **W** → **W** → **W**

**divide & conquer (fixed degree):**

1. 2
2. 5 13 6 9 14 10
3. 7 11 15
4. 8 12 16

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

Master-Worker:

Google Map-Reduce:

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

1. The Language Eden (in a nutshell)
2. Skeleton-Based Programming
3. Small-Scale Skeletons: Map and Reduce
4. Process Topologies as Skeletons
5. Algorithm-Oriented Skeletons: Two Classics
6. Summary
Basic Skeletons: Higher-Order Functions

- **Parallel transformation:** Map
  
  \[
  \text{map} :: (a \to b) \to [a] \to [b]
  \]

  independent elementwise transformation

  ... probably the most common example of parallel functional programming (called "embarassingly parallel")
Basic Skeletons: Higher-Order Functions

- **Parallel transformation**: Map
  \[
  \text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
  \]
  independent elementwise transformation
  ...probably the most common example of parallel functional programming (called "embarassingly parallel")

- **Parallel Reduction**: Fold
  \[
  \text{fold} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a
  \]
  with commutative and associative operation.

- **Parallel Scan**:
  \[
  \text{parScanL} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a]
  \]
  reduction keeping the intermediate results.

- **Parallel Map-Reduce**:
  combining transformation and groupwise reduction.
Embarassingly Parallel: \texttt{map}

\texttt{map}: apply \texttt{transformation} to all elements of a list

- Straight-forward element-wise parallelisation

\[
\texttt{parmap} :: (\text{Trans } a, \text{Trans } b) \Rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\texttt{parmap} = \texttt{spawn} \ . \ \texttt{repeat} \ . \ \texttt{process} \\
\texttt{parmap } f \ \texttt{xs} = \texttt{spawn} \ (\texttt{repeat} \ (\texttt{process} \ f)) \ \texttt{xs}
\]
Embarassingly Parallel: map

map: apply transformation to all elements of a list

- Straight-forward element-wise parallelisation

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

  Much too fine-grained!
Embarassingly Parallel: map

map: apply transformation to all elements of a list

- Straight-forward element-wise parallelisation
  
  \[
  \text{parmap} :: \text{(Trans } a, \text{ Trans } b) \Rightarrow \text{ (} a \rightarrow b \text{) } \rightarrow \text{ [} a \text{] } \rightarrow \text{ [} b \text{]}
  \]
  
  \[
  \text{parmap} = \text{spawn} \cdot \text{repeat} \cdot \text{process}
  \]
  
  **-- parmap f xs = spawn (repeat (process f)) xs**

  Much too fine-grained!

- Group-wise processing: Farm of processes

  \[
  \text{farm} :: \text{(Trans } a, \text{ Trans } b) \Rightarrow \text{ (} a \rightarrow b \text{) } \rightarrow \text{ [} a \text{] } \rightarrow \text{ [} b \text{]}
  \]
  
  \[
  \text{farm f xs} = \text{join results}
  \]
  
  **where results = spawn (repeat (process (map f))) parts**
  
  \[
  \text{parts} = \text{distribute noPe xs} -- \text{noPe, so use all nodes}
  \]
  
  \[
  \text{join} = \ldots
  \]
  
  \[
  \text{distribute n =} \ldots -- \text{join} \cdot \text{distribute n} == \text{id}
  \]
Example

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

Mandelbrot (Pseudocode)

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

---

![Mandelbrot Set Image]
Example / Exercise

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

Mandelbrot (Pseudocode)

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

Exercise:

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

Framework programs can be found on the course pages...
Example / Exercise: Chunked Tasks

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

**Mandelbrot (Pseudocode)**

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

Slide 17/36 — J.Berthold — Eden — Heriot-Watt, 03/2013
Example / Exercise: Chunked Tasks

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

**Mandelbrot (Pseudocode)**

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

Simple chunking leads to load imbalance (task complexities differ)
Example / Exercise: Round-robin Tasks

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

Mandelbrot (Pseudocode)

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

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

Worker nodes transform elementwise:
worker :: task -> result

Master node manages task pool
mw :: Int -> Int ->
    ( a -> b ) -> [a] -> [b]
mw np prefetch f tasks = ...

Parameters: no. of workers, prefetch

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

Worker nodes transform elementwise:
worker :: task -> result

Master node manages task pool
mw :: Int -> Int ->
    ( a -> b ) -> [a] -> [b]
mw np prefetch f tasks = ...

Parameters: no. of workers, prefetch

- Master sends a new task each time a result is returned
  (needs many-to-one communication)

- Initial workload of \texttt{prefetch} tasks for each worker:
  Higher prefetch \Rightarrow more and more static task distribution
  Lower prefetch \Rightarrow dynamic load balance

- Result order needs to be reestablished!
Master-Worker: An Implementation

Master-Worker Skeleton Code

```haskell
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

```haskell
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 ++ newReq
  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

- \textbf{foldl} :: (b \to a \to b) \to b \to [a] \to b
- \textbf{foldr} :: (a \to b \to b) \to b \to [a] \to b

Starting from the \textbf{left} or \textbf{right}, implying different reduction function.

- To parallelise: break into sublists and pre-reduce in parallel.
- Better options if order does not matter.
Parallel Reduction, Map-Reduce

Reduction ($\text{fold}$) usually has a direction

- $\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

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)

```haskell
sumEuler
result = foldl (+) 0 (map phi [1..n])
phi k = length (filter (\ n -> gcd n k == 1) [1..(k-1)])
```
Parallel Map-Reduce: Restrictions

- `parmapReduceStream :: Int -> (a -> b) -> (b -> b -> b) -> b -> [a] -> b`

  `parmapReduceStream np mapF redF neutral list = foldl redF neutral subRs`

  where sublists = `distribute np list`

  subFold = `process (foldl' redF neutral . (map mapF))`

  subRs = `spawn (replicate np subFold) sublists`
Parallel Map-Reduce: Restrictions

- \texttt{parmapReduceStream} :: Int \to
  \quad (a \to b) \to (b \to b \to b) \to b \to
  \quad [a] \to b

\begin{verbatim}
parmapReduceStream np mapF redF neutral list = foldl redF neutral subRs
  where sublists = distribute np list
       subFold = process (foldl’ redF neutral \cdot (map mapF))
       subRs   = spawn (replicate np subFold) sublists
\end{verbatim}

- Associativity and \textbf{neutral element} (essential).
- \textbf{commutativity} (desired, more liberal distribution)
- \textbf{need to narrow type} of the reduce parameter function!
- \ldots\textbf{Alternative fold type}: \texttt{redF’} :: [b] \to b

\begin{verbatim}
redF’ []  = neutral
redF’ (x:xs) = foldl’ redF x xs
\end{verbatim}
Google Map-Reduce: Grouping Before Reduction

\[
g\text{MapRed} :: (k_1 \rightarrow v_1 \rightarrow [(k_2, v_2)]) \quad -- \text{mapF}
\rightarrow (k_2 \rightarrow [v_2] \rightarrow \text{Maybe } v_3) \quad -- \text{reduceF}
\rightarrow \text{Map } k_1 \text{ v1 } \rightarrow \text{Map } k_2 \text{ v3} \quad -- \text{input / output}
\]

1. Input: key-value pairs \((k_1, v_1)\), many or no outputs \((k_2, v_2)\)
2. Intermediate grouping by key \(k_2\)
3. Reduction per (intermediate) key \(k_2\) (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)


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
Outline

1. The Language Eden (in a nutshell)
2. Skeleton-Based Programming
3. Small-Scale Skeletons: Map and Reduce
4. Process Topologies as Skeletons
5. Algorithm-Oriented Skeletons: Two Classics
6. Summary
Process Topologies as Skeletons: Explicit Parallelism

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

Examples:

- **Pipeline/Ring:**
- **Master/Worker:**
- **Hypercube:**
Process Topologies as Skeletons: Explicit Parallelism

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

**Examples:**

- Pipeline/Ring:
- Master/Worker:
- Hypercube:

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

![Ring Skeleton Diagram]

```haskell
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
Outline

1. The Language Eden (in a nutshell)
2. Skeleton-Based Programming
3. Small-Scale Skeletons: Map and Reduce
4. Process Topologies as Skeletons
5. Algorithm-Oriented Skeletons: Two Classics
6. Summary
Two Algorithm-oriented Skeletons

- **Divide and conquer**

  \[
  \text{divCon} :: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \quad -- \text{trivial? / then solve}
  \]

  \[
  \rightarrow (a \rightarrow [a]) \rightarrow ([b] \rightarrow b) \quad -- \text{split / combine}
  \]

  \[
  \rightarrow a \rightarrow b \quad -- \text{input / result}
  \]

- **Iteration**

  \[
  \text{iterateUntil} :: \text{(inp} \rightarrow ([ws],[t],ms)) \rightarrow \quad -- \text{split/init function}
  \]

  \[
  (t \rightarrow \text{State ws r}) \rightarrow \quad -- \text{worker function}
  \]

  \[
  ([r] \rightarrow \text{State ms (Either out [t]))} \rightarrow \quad -- \text{manager function}
  \]

  \[
  \rightarrow \text{inp} \rightarrow \text{out}
  \]
Divide and Conquer Skeletons

- General version: no assumptions on problem characteristics
  
  \[
  \text{divCon} :: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \quad \text{-- trivial? / then solve}
  \]
  
  \[
  \rightarrow (a \rightarrow [a]) \rightarrow ([b] \rightarrow b) \quad \text{-- split / combine}
  \]
  
  \[
  \rightarrow a \rightarrow b \quad \text{-- input / result}
  \]

  \[
  \text{divCon} \text{ trivial solve split combine} = \ldots
  \]

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

- General version: no assumptions on problem characteristics

  \[
  \text{divCon} :: (a \to \text{Bool}) \to (a \to b) \quad \text{-- trivial? / then solve}
  \to (a \to [a]) \to ([b] \to b) \quad \text{-- split / combine}
  \to a \to b \quad \text{-- input / result}
  \]
  \[
  \text{divCon trivial solve split combine} = \ldots \quad \text{-- you write one}
  \]

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

Exercise:

- Implement this general divide-and-conquer version.
  Write a sequential version first, then make recursive calls parallel. Add one \text{Int} parameter to limit the parallel depth.
Iteration Skeleton

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

```
iterateUntil :: (inp -> ([ws],[t],ms)) ->
          (t -> State ws r) ->
          ([r] -> State ms (Either out [t])) ->
          inp -> out
```

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

**Manager:** decides whether to continue,
based on master state `ms` and all worker results.
produce tasks for all workers
Outline

1. The Language Eden (in a nutshell)
2. Skeleton-Based Programming
3. Small-Scale Skeletons: Map and Reduce
4. Process Topologies as Skeletons
5. Algorithm-Oriented Skeletons: Two Classics
6. Summary
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, ...)
  - 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, …)
  - 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/)
Exercises for the Lab

1. Complete the Hamming number program
   File: hamming-.hs
   Execute the program and look at an execution trace using EdenTV

2. Implement two versions of parMap which increase granularity
   Files: ParMap.hs, mandel.hs
   Test your versions using the Mandelbrot program.

3. Implement the Divide-And-Conquer skeleton
   Files: DC.hs, mergesort.hs
   Test your skeleton implementation using the provided mergesort program.

4. (Bonus) Implement a simple quicksort program using the skeleton
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