Dusty Decks of parallel Haskell

– for the lack of a catchier title–

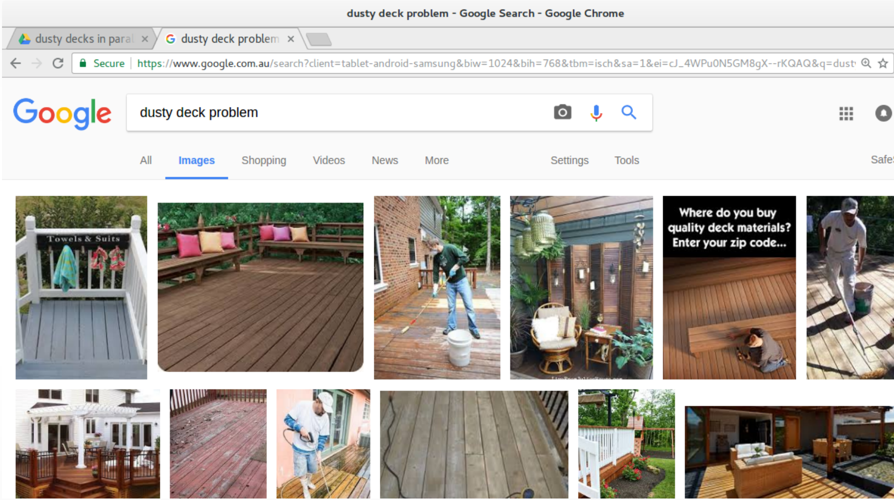
Jost Berthold[*][‡]

jberthold@acm.org

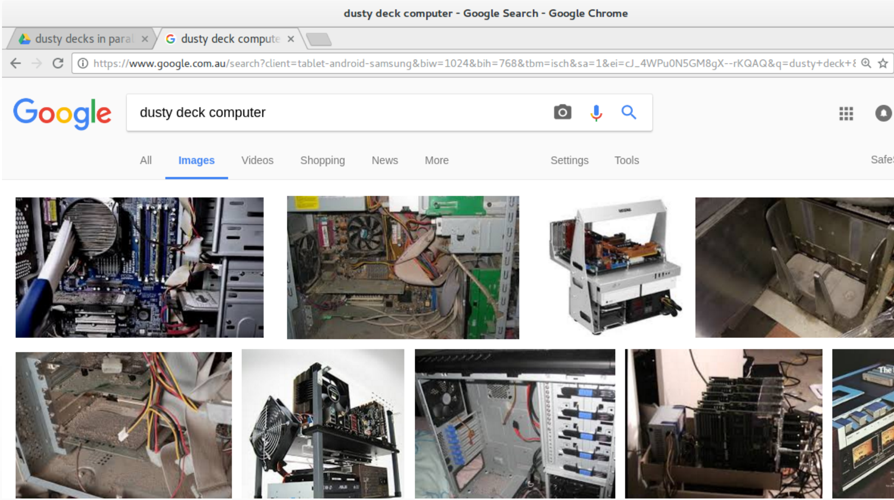
Commonwealth Bank
(but for this topic it is probably more accurate to say
Philipps University of Marburg, Germany)

FP Syd, April 2017

Dusty ..WHAT?..


dusty deck problem - Google Search - Google Chrome
dusty decks in paral
dusty deck problem
Secure
https://www.google.com.au/search?client=tablet-android-samsung&biw=1024&bih=768&tbm=isch&sa=1&ei=cJ_4WPu0N5GM8gX--rKQAQ&q=dust
Google
dusty deck problem
All
Images
Shopping
Videos
News
More
Settings
Tools
Towels & Suits
Where do you buy quality deck materials? Enter your zip code...

Dusty ..WHAT?.. but something with computers



The 'Dusty deck' problem in parallel computing

Programming Multicores: Do Applications Programmers Need to Write Explicitly Parallel Programs?

Arvind
Massachusetts Institute of Technology

David August
Princeton University

Keshav Pingali
Derek Chiou
University of Texas at Austin

Resit Sendag
University of Rhode Island

Joshua J. Yi

In this panel discussion from the 2009 Workshop on Computer Architecture Research Directions, David August and Keshav Pingali debate whether explicitly parallel programming is a necessary evil for applications programmers, assess the current state of parallel programming models, and discuss possible routes toward finding the programming model for the multicore era.

Moderator's introduction: Arvind

Do applications programmers need to write explicitly parallel programs? Most people believe that the current method of parallel programming is impeding the exploitation of multicores. In other words, the number of cores in a microprocessor is likely to track Moore's law in the near future, but the programming of multicores might remain the biggest obstacle in the forward march of performance.

Let's assume that this premise is true. Now, the real question becomes: how should applications programmers exploit the potential of multicores? There have been two main

in the 1970s and 1980s, when two main approaches were developed.

The first approach required that the compilers do all the work in finding the parallelism. This was often referred to as the "dusty decks" problem—that is, how to exploit parallelism in existing programs. This approach taught us a lot about compiling. But most importantly, it taught us how to write a program in the first place, so that the compiler had a chance of finding the parallelism.

The second approach, to which I also contributed, was to write programs in a manner such that the inherent (or obvious) parallelism in the algorithm is not obscured in the

in: *IEEE Micro*, vol.30 no.3, pp.19-33, May 2010

The 'Dusty deck' problem in parallel computing

PROGRAMMING MULTICORES: DO APPLICATIONS PROGRAMMERS NEED TO WRITE EXPLICITLY PARALLEL PROGRAMS?

Arvind
Institute
chnology
August
niversity
Pingali
k Chiou
of Texas
t Austin
Sendag
f Rhode
Island

IN THIS PANEL DISCUSSION FROM THE 2009 WORKSHOP ON COMPUTER ARCHITECTURE RESEARCH DIRECTIONS, DAVID AUGUST AND KESHAV PINGALI DEBATE WHETHER EXPLICITLY PARALLEL PROGRAMMING IS A NECESSARY EVIL FOR APPLICATIONS PROGRAMMERS, ASSESS THE CURRENT STATE OF PARALLEL PROGRAMMING MODELS, AND DISCUSS POSSIBLE ROUTES TOWARD FINDING THE PROGRAMMING MODEL FOR THE MULTICORE ERA.

Moderator's introduction: Arvind

Do applications programmers need to write explicitly parallel programs? Most people believe that the current method of parallel programming is impeding the exploitation of multicores. In other words, the number of cores in a microprocessor is likely to track Moore's law in the near future, but the programming of multicores might remain the biggest obstacle in the forward march of performance.

Let's assume that this premise is true. Now, the real question becomes: how should applications programmers exploit the potential of multicores? There have been two main

in the 1970s and 1980s, when two main approaches were developed.

The first approach required that the compilers do all the work in finding the parallelism. This was often referred to as the "dusty decks" problem—that is, how to exploit parallelism in existing programs. This approach taught us a lot about compiling. But most importantly, it taught us how to write a program in the first place, so that the compiler had a chance of finding the parallelism.

The second approach, to which I also contributed, was to write programs in a manner such that the inherent (or obvious) parallelism in the algorithm is not obscured in the program. I explored declarative languages for

in: *IEEE Micro*, vol.30 no.3, pp.19-33, May 2010

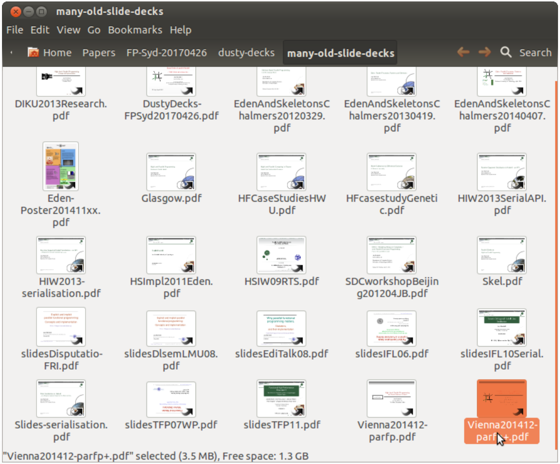
'Dusty decks' in this talk

Classic *"dusty deck"*

'Dusty decks' in this talk

Classic *"dusty deck"*

Jost's dusty decks

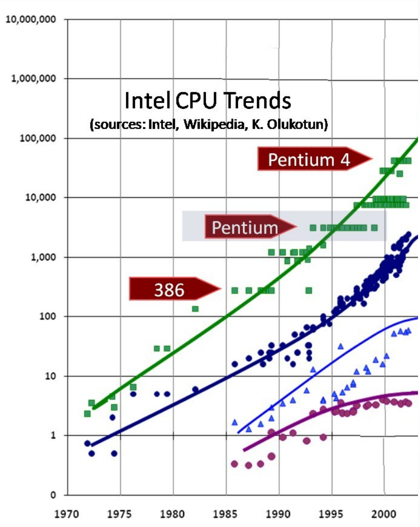


Overview

1. Prelude on dusty decks
2. A few things on parallel programming
3. Eden, a parallel Haskell for distributed memory
4. Skeletons for parallel programming: A Selection
   - Topology Skeletons – and a lesson about strictness
   - Hello-world of parallel FP: **maps** and beyond (task pools)
   - Algorithmic (higher-level) skeletons
5. Some conclusions

Overview

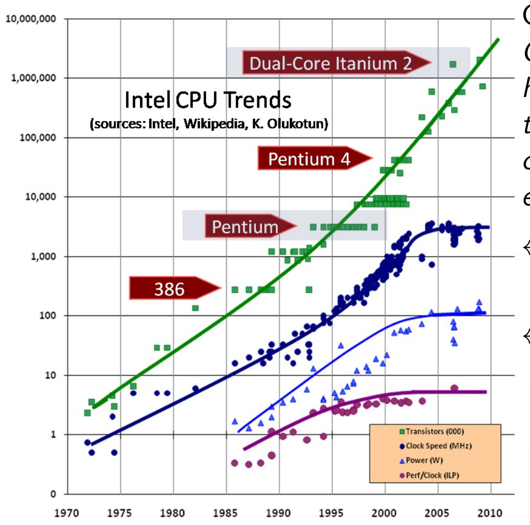
Why we care about parallel programming



Gordon Moore, 1965:
Over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years.

[?]

Why we care about parallel programming



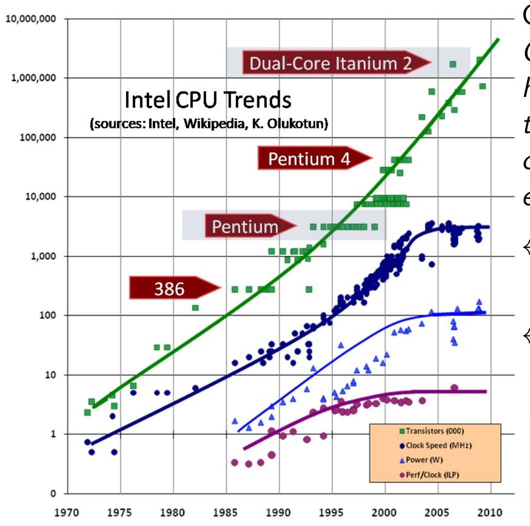
Gordon Moore, 1965:
Over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years.

← Not so for clock speeds!

← (nor power consumption)

[?]

Why we care about parallel programming

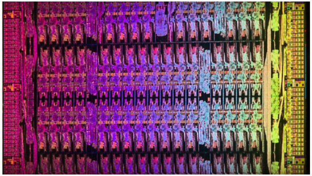


[?]

Gordon Moore, 1965:
Over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years.

← Not so for clock speeds!

← (nor power consumption)



Parallel programming is cumbersome

- Shared data needs to be protected (locks)
  - protection can lead to deadlocks,
  - omitting it can lead to race conditions.
- Relaxed memory consistency of the hardware can falsify reasonable assumptions of the programmer
- (Point-to-point) message passing is error-prone and relies on complex assumptions about send/receive (a-)synchronicity.

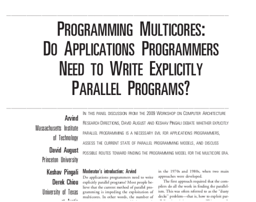
Algorithm and essential complexity are often buried in gory details.

Parallel functional programming operates at a higher abstraction level: Problem decomposition, task granularity, data dependencies

Explicit and implicit parallel programming

Summary of the debate

- Camps of implicit vs. explicit parallel programming
  - regular and fine-grained vs.
  - amorphous, coarse-grained, and input-dependent
- programming happens at different levels
- no final answer



Programming Multicores: Do Applications Programmers Need to Write Explicitly Parallel Programs?

Arvind
Massachusetts Institute of Technology
David August
Princeton University
Keshav Pingali
Derek Chiou
University of Texas

In this panel discussion from the 2009 Workshop on Computer Architecture Research Directions, David August and Keshav Pingali debate whether explicitly parallel programming is a necessary evil for applications programmers, assess the current state of parallel programming models, and discuss possible routes toward finding the programming model for the multicore era.

Moderator's introduction: Arvind

Do applications programmers need to write explicitly parallel programs? Most people believe that the current method of parallel programming is impeding the exploitation of multicores. In other words, the number of

in the 1970s and 1980s, when two main approaches were developed.

The first approach required that the compilers do all the work in finding the parallelism. This was often referred to as the "dusty decks" problem—that is, how to exploit par-

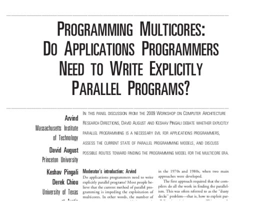
Explicit and implicit parallel programming

Summary of the debate

- Camps of implicit vs. explicit parallel programming
  - regular and fine-grained vs.
  - amorphous, coarse-grained, and input-dependent
- programming happens at different levels
- no final answer

Similar questions in the functional space:

- How much abstraction and automation is useful (for which application profile)?
- How much explicit control is required for performance?



PROGRAMMING MULTICORES: DO APPLICATIONS PROGRAMMERS NEED TO WRITE EXPLICITLY PARALLEL PROGRAMS?

Arvind
Massachusetts Institute of Technology
David August
Princeton University
Keshav Pingali
Derek Chiou
University of Texas

IN THIS PANEL DISCUSSION FROM THE 2009 WORKSHOP ON COMPUTER ARCHITECTURE RESEARCH DIRECTIONS, DAVID AUGUST AND KESHAV PINGALI DEBATE WHETHER EXPLICITLY PARALLEL PROGRAMMING IS A NECESSARY EVIL FOR APPLICATIONS PROGRAMMERS, ASSESS THE CURRENT STATE OF PARALLEL PROGRAMMING MODELS, AND DISCUSS POSSIBLE ROUTES TOWARD FINDING THE PROGRAMMING MODEL FOR THE MULTICORE ERA.

Moderator's introduction: Arvind

Do applications programmers need to write explicitly parallel programs? Most people believe that the current method of parallel programming is impeding the exploitation of multicores. In other words, the number of

in the 1970s and 1980s, when two main approaches were developed.

The first approach required that the compilers do all the work in finding the parallelism. This was often referred to as the "dusty decks" problem—that is, how to exploit par-

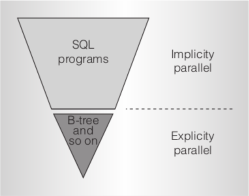
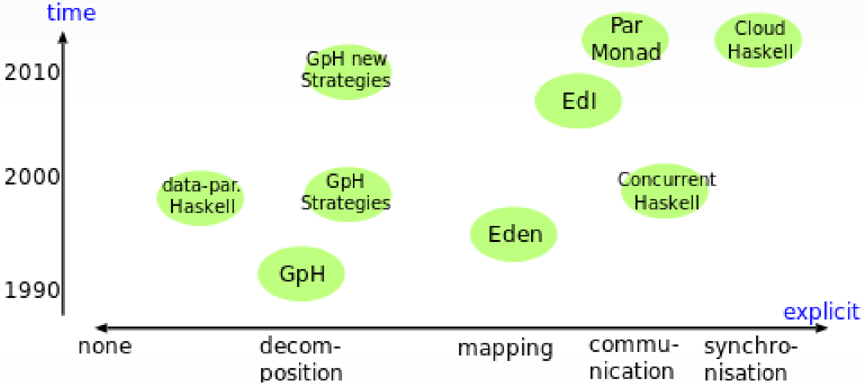


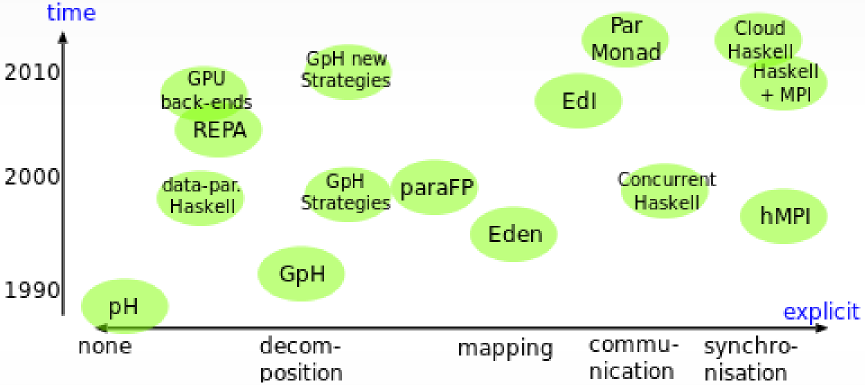
Figure 7. A framework for understanding the implicitly and explicitly parallel programming models. Applications programmers write SQL programs, which are implicitly parallel, and they rely on implementations of relations, such as B-trees, which systems programmers have carefully coded in parallel.

Explicit and implicit. . . : A landscape



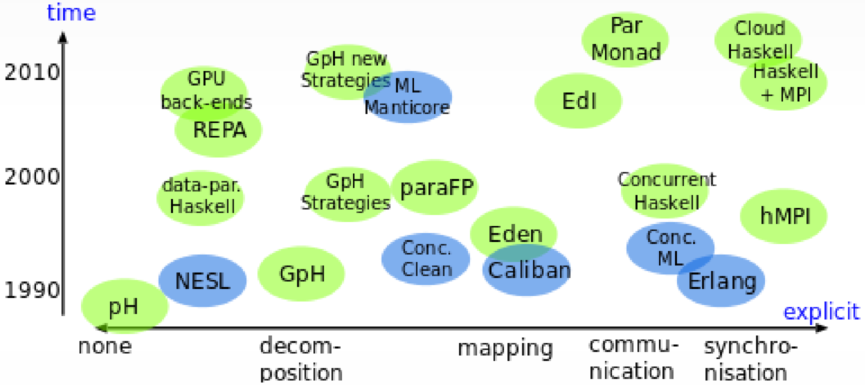
Classification of parallel programming paradigms (inspired by D.Skillikorn)

Explicit and implicit. . . : A landscape



Classification of parallel programming paradigms (inspired by D.Skillikorn)

Explicit and implicit. . . : A landscape



Classification of parallel programming paradigms (inspired by D.Skillikorn)

Overview

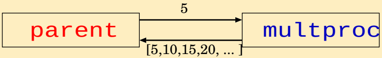
Eden Examples in Pictures

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

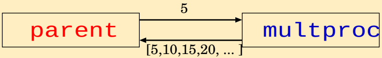
Eden Examples in Pictures

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

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

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

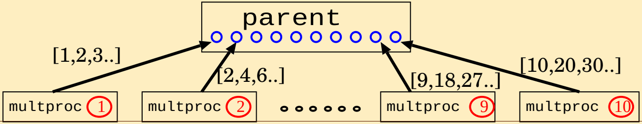
```
multiple5 = multproc # 5
```



- 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]
```



Eden: Explicit Parallel Evaluation

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

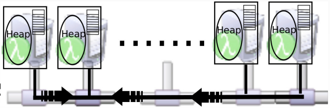
Eden constructs for Process abstraction and instantiation

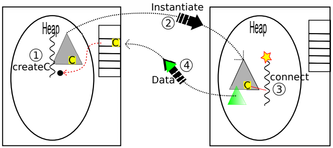
```
process ::(Trans a, Trans b)=> (a -> b) -> Process a b
( # ) :: (Trans a, Trans b) => (Process a b) -> a -> b
spawn :: (Trans a, Trans b) => [ Process a b ] -> [a] -> [b]
```

- 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

Eden implementation

- Explicit message passing between independent runtime system instances
- Interface to Haskell: IO-monadic primitive operations
- Haskell module for functional API (process, instantiation)



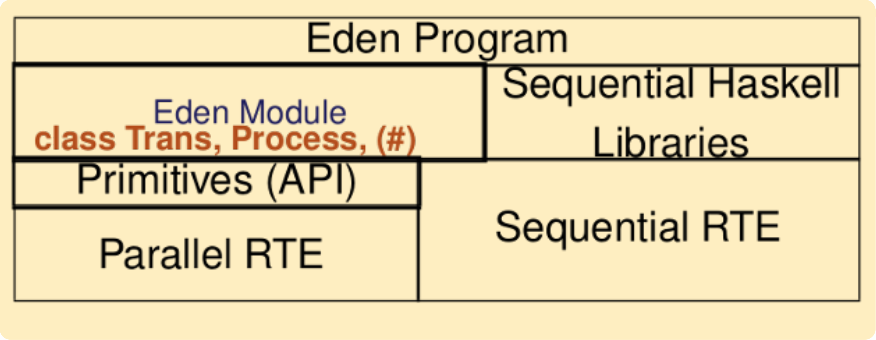



```
instantiateAt :: Int -> Process a b -> a -> IO b
instantiateAt p (Proc f_remote) procInput = do
         (sendResult,  r  )      <- createComm  -- result communicator
         (inCC, Comm sendInput) <- createC     -- reply: input communicator
         sendData (Instantiate p) (f_remote sendResult inCC)
         fork (sendInput procInput)
         return r

{-# NOINLINE ( # ) #-}
p # x = unsafePerformIO $ instantiateAt 0 p x
```

Implementation layers

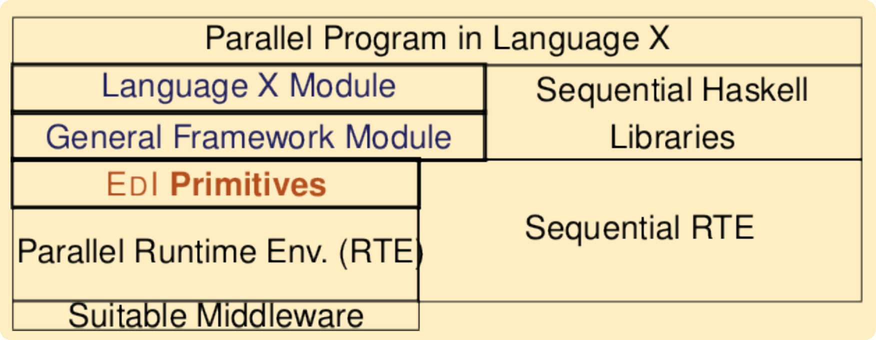
In line with the earlier question of required control:



- Where should the line be drawn between pure and impure code?

Implementation layers

In line with the earlier question of required control:



- Where should the line be drawn between pure and impure code?
- . . . and libraries are not even in the picture

Overview

The Idea of Skeleton-Basked Parallelism

How much code do you need to implement a parallel quick sort?

The Idea of Skeleton-Basked Parallelism

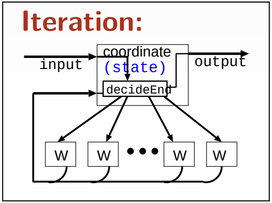
How much code do you need to implement a parallel quick sort?

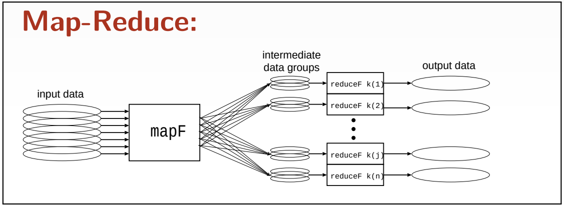
Divide and Conquer, as a higher-order function

```
divConqB :: (a -> Bool)          -- trivial?
         -> (a -> b)             -- solve
         -> (a -> [a])           -- split
         -> (a -> [b] -> b)      -- combine
         -> a -> b
divConqB trivial solve divide combine input = ...
```

- Higher-order function defines algorithmic structure
- Parameter functions define concrete algorithm
- Parallel structure (binary tree) can be exploited for parallelism

Parallel Data Processing Using Parallel Skeletons

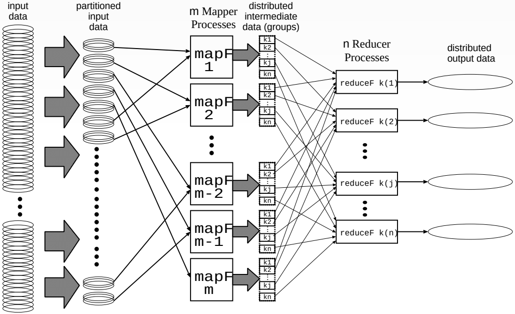


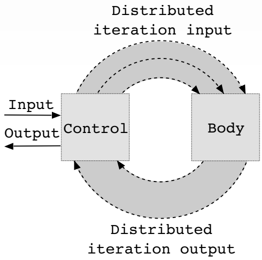


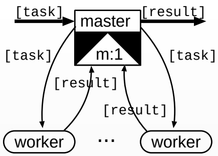
- Parallel Skeletons [Cole 1989]: abstract specification of. . .
- . . . algorithm structure as a higher-order function.
- Abstract over concrete tasks (embedded "worker" functions),
- hidden parallel optimised implementation(s) (machine-specific)

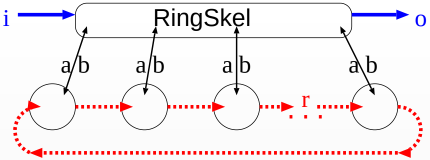
Enable a high-level view on parallel systems and computations

Parallel Data Processing Using Parallel Skeletons







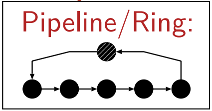


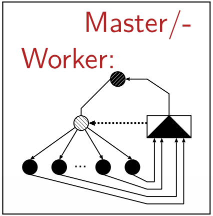
Topology Skeletons – and a lesson about strictness

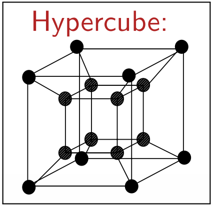
Process Topologies as Skeletons: Explicit Parallelism

- Parallel interaction of a process structure described as a pattern/higher-order function
- Node behaviour defined as function argument, skeleton structures parallel interaction;

Examples:





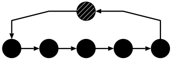


Process Topologies as Skeletons: Explicit Parallelism

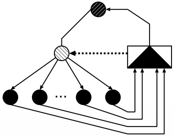
- Parallel interaction of a process structure described as a pattern/higher-order function
- Node behaviour defined as function argument, skeleton structures parallel interaction;

Examples:

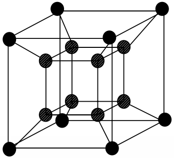
Pipeline/Ring:



Master/-Worker:



Hypercube:



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

- Explicit notion of parallelism and communication;
- capitalises on structured methodology and portability.

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?

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)

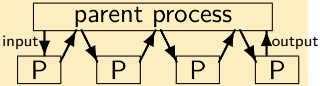
Restricting to stages homogenous by their types

```
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)
```

Purely Functional Pipeline? (a topology skeleton)

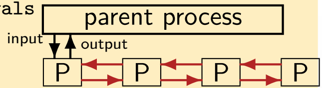
Restricting to stages homogenous by their types

```
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)

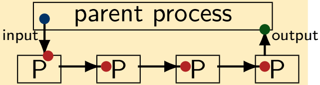
Recursion with dynamic reply channel:

```
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)
```

Pipeline (cont.d)

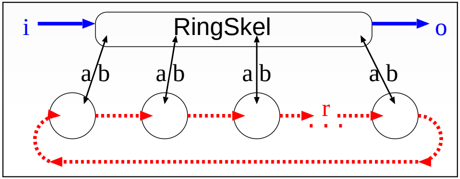
Recursion with dynamic reply channel:

```
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).

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 = ...
```

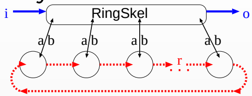
- Circulating global data between worker nodes (stream of type `[r]`)
- All ring processes connect to parent to receive input/send output
- Parameters: functions for
  - decomposing input, combining output, ring worker

Ring Example: All Pairs Shortest Paths (Floyd-Warshall)

Adjacency Matrix | Distance Matrix

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

- For each row of distances from node k:
  - For all other distance rows i, in ascending order:
  - check if row i indicates a path from k to another node
  - if yes, update the distance row k to use the shorter path
- When row k has been updated with all $i < k$
  - use this updated distance row to update all rows $j > k$.
- Order of updates matters, but all rows can be updated for each i simultaneously.



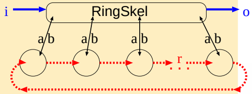
Ring Example: All Pairs Shortest Paths (Floyd-Warshall)

Adjacency Matrix — Distance Matrix

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

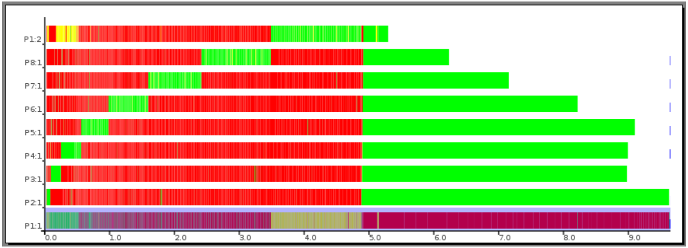
Floyd-Warshall: Update all rows k in parallel

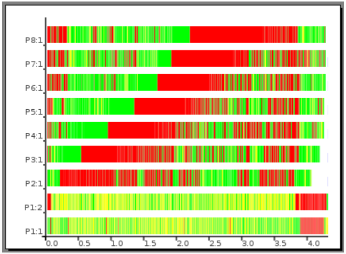
```
ring_iterate :: Int -> Int -> Int -> [Int] -> [[Int]] -> ([Int],[[Int]])
ring_iterate size k i rowk rows
    | i > size =  (rowk, [])            -- finished
    | i == k   =  (result, rowk:rest) -- send own row
    | otherwise = (result, rowi:rest)
    where rowi:xs          = rows
          (result, rest) = ..???.. ring_iterate size k (i+1) nextrowk xs
          nextrowk | i == k     = rowk
                   | otherwise = updaterow rowk rowi distki
          distki           = rowk!!(i-1)
```



Trace of Warshall Program

First version:





With additional early demand

```
    ...
(result, rest) = rnf nextrowk `seq`
                    ring_iterate size k (i+1) nextrowk xs
nextrowk | i == k     = rowk
         | otherwise = updaterow rowk rowi distki
    ...
```

Hello-world of parallel FP: maps and beyond (task pools)

Small-Scale Skeletons: Higher-Order Functions

- Parallel transformation: Map

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

independent, elementwise, embarrassingly parallel

…probably the most common example of parallelism in FP

- 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.

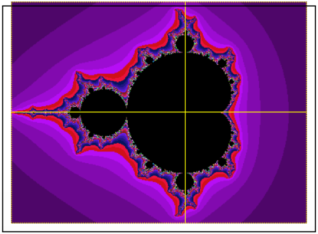
- Parallel Map-Reduce:

combining transformation and reduction.

Parallel `map` Example: Mandelbrot

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

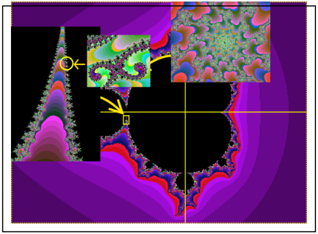
```
pic :: ..picture-parameters.. -> PPMAscii
pic threshold ul lr dimx np s = ppmheader ++ concat (parMap computeRow rows)
   where rows = ...dimx..ul..lr..
         parMap = ...
```



Parallel `map` Example: Mandelbrot

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

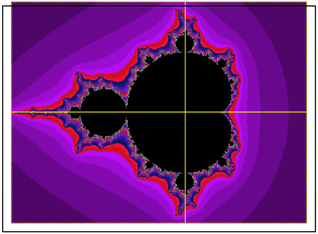
```
pic :: ..picture-parameters.. -> PPMAscii
pic threshold ul lr dimx np s = ppmheader ++ concat (parMap computeRow rows)
   where rows = ...dimx..ul..lr..
         parMap = ...
```

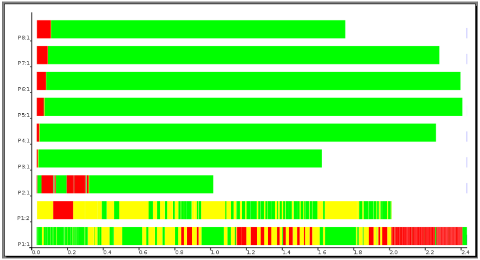


Parallel map Example: Mandelbrot

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

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



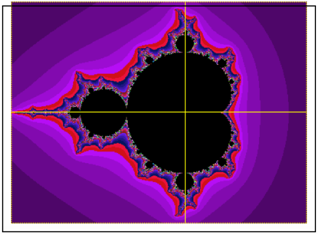


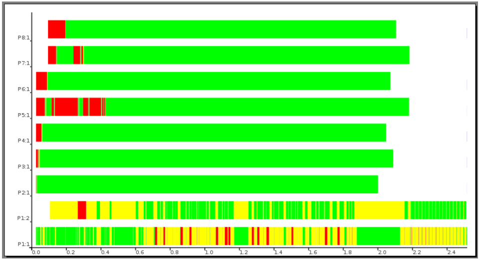
Very uneven load balance when using chunks (stripes)

Parallel `map` Example: Mandelbrot

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

```
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.

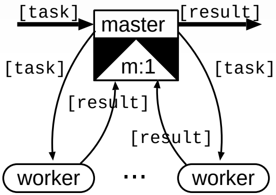
Dynamic Load-Balancing: Master-Worker Skeleton

Worker nodes transform elementwise:

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

Master node manages task pool

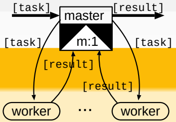
```
mw  :: Int -> Int ->
       ( a -> b ) -> [a] -> [b]
mw np prefetch f tasks =  ...
```



Parameters: no. of workers, prefetch

- Master sends a new task each time a result is returned
- Initial task `prefetch` for each worker:
  Higher prefetch ⇒ more and more static task distribution
  Lower prefetch ⇒ dynamic load balance

Workpool skeleton (simple version)



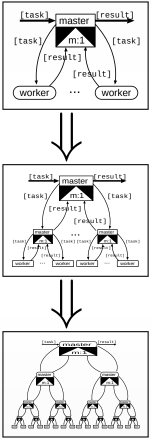
Simple Workpool Skeleton

```
mw np prefetch f tasks = map snd fromWorkers
  where fromWorkers :: [(Int,r)]
        fromWorkers = merge (tagWithIds (parMapEden (map f) taskss))
        taskss      = distribute (initialReqs ++ newReqs) tasks
        initialReqs = concat (replicate prefetch [1..np])
        newRequests = map fst fromWorkers
        distribute :: [Int] -> [t] -> [[t]]
        distribute reqs tasks = [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 _      _      _  = []
        tagWithIds rss = [ zip (repeat i) rs |(i,rs) <-zip [1..] rss]
```

- Non-deterministic (unsorted results), implemented using `merge`
- Returned results tagged, driving task distribution
- Many variants available in the Eden skeleton library.

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

Nesting the Master-Worker Skeleton

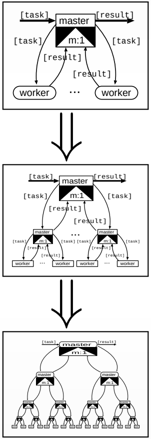


- Nesting: . . . We use a "more practical" interface:

```
mw' :: Int -> Int -> ([a] -> [b]) -> [a] -> [b]
```

Skeleton and worker function now have the same type!

Nesting the Master-Worker Skeleton



- Nesting: . . . We use a "more practical" interface:

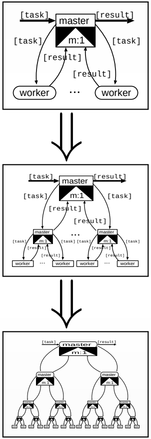
```
mw' :: Int -> Int -> ([a] -> [b]) -> [a] -> [b]
```

  Skeleton and worker function now have the same type!

- 2-Level Nesting:

```
mw2 np1 pf1  np2 pf2 wf = mw' np1 pf1 (mw' np2 pf2 wf)
```

Nesting the Master-Worker Skeleton



- Nesting: ... We use a "more practical" interface:

```
mw' :: Int -> Int -> ([a] -> [b]) -> [a] -> [b]
```

Skeleton and worker function now have the same type!

- 2-Level Nesting:

```
mw2 np1 pf1  np2 pf2 wf = mw' np1 pf1 (mw' np2 pf2 wf)
```

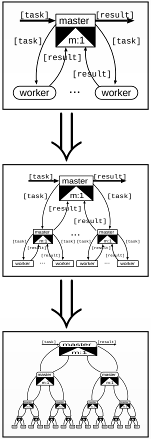
- General nesting by folding:

```
fld :: (Trans t, Trans r) => (Int,Int) -> ([t]->[r]) -> ([t]->[r])
fld (np,pf) wf = mw' np pf wf
```

Branch degrees `nps` and prefetch values `pfs` per level

```
mwNested nps pfs wf = foldr fld wf (zip nps pfs)
```

Nesting the Master-Worker Skeleton



- Nesting: ...We use a "more practical" interface:

```
mw' :: Int -> Int -> ([a] -> [b]) -> [a] -> [b]
```

Skeleton and worker function now have the same type!

- 2-Level Nesting:

```
mw2 np1 pf1  np2 pf2 wf = mw' np1 pf1 (mw' np2 pf2 wf)
```

- General nesting by folding:

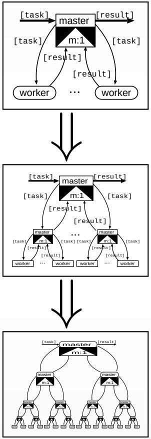
```
fld :: (Trans t, Trans r) => (Int,Int) -> ([t]->[r]) -> ([t]->[r])
fld (np,pf) wf = mw' np pf wf
```

Branch degrees `nps` and prefetch values `pfs` per level

```
mwNested nps pfs wf = foldr fld wf (zip nps pfs)
```

What can possibly go wrong?

Nesting the Master-Worker Skeleton



- Nesting: ... We use a "more practical" interface:

```
mw' :: Int -> Int -> ([a] -> [b]) -> [a] -> [b]
```

Skeleton and worker function now have the same type!

- 2-Level Nesting:

```
mw2 np1 pf1  np2 pf2 wf = mw' np1 pf1 (mw' np2 pf2 wf)
```

- General nesting by folding:

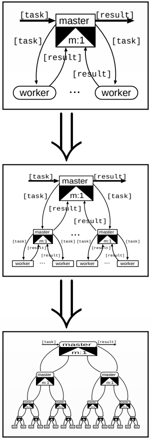
```
fld :: (Trans t, Trans r) => (Int,Int) -> ([t]->[r]) -> ([t]->[r])
fld (np,pf) wf = mw' np pf wf
```

Branch degrees `nps` and prefetch values `pfs` per level

```
mwNested nps pfs wf = foldr fld wf (zip nps pfs)
```

What can possibly go wrong? ......

Nesting the Master-Worker Skeleton



- Nesting: . . . We use a "more practical" interface:

```
mw' :: Int -> Int -> ([a] -> [b]) -> [a] -> [b]
```

Skeleton and worker function now have the same type!

- 2-Level Nesting:

```
mw2 np1 pf1  np2 pf2 wf = mw' np1 pf1 (mw' np2 pf2 wf)
```

- General nesting by folding:

```
fld :: (Trans t, Trans r) => (Int,Int) -> ([t]->[r]) -> ([t]->[r]
fld (np,pf) wf = mw' np pf wf
```

Branch degrees `nps` and prefetch values `pfs` per level

```
mwNested nps pfs wf = foldr fld wf (zip nps pfs)
```

What can possibly go wrong? . . . . . . `wf = drop prefetch` †

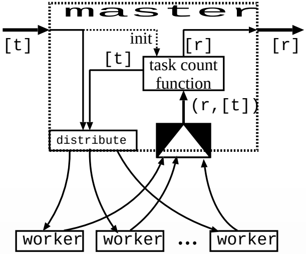
Dynamically Growing Task Pools

- More interesting: `worker :: task -> (Maybe result,[task])`
- New tasks enqueued in dynamically growing task pool.
- Backtracking: Explore decision alternatives until desired result.

Dynamically Growing Task Pools

- More interesting: `worker :: task -> (Maybe result,[task])`
- New tasks enqueued in dynamically growing task pool.
- Backtracking: Explore decision alternatives until desired result.

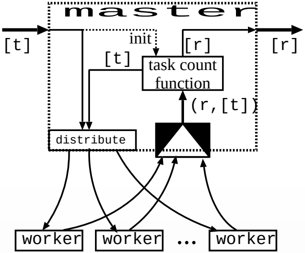
- State: Counter for total no. of tasks
- Task counter function:
  - consumes output of all workers
  - adds new tasks to task list
  - closes task list when counter $==0$



Dynamically Growing Task Pools

- More interesting: `worker :: task -> (Maybe result,[task])`
- New tasks enqueued in dynamically growing task pool.
- Backtracking: Explore decision alternatives until desired result.

- State: Counter for total no. of tasks
- Task counter function:
  - consumes output of all workers
  - adds new tasks to task list
  - closes task list when counter == 0



- This is a computation scheme, rather than being data-oriented.

Algorithmic (higher-level) skeletons

More algorithm-oriented Skeletons

Backtracking (Tree search)

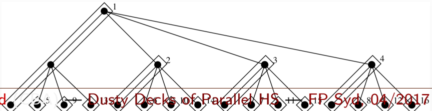
```
backtrack :: (a -> (Maybe b, [a])  -- maybe solve problem, refine problem
          -> a -> [b]              -- start problem / solutions
```

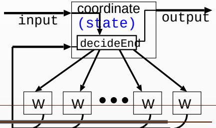
Divide and conquer

```
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
```





Divide & Conquer (simple general version)

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

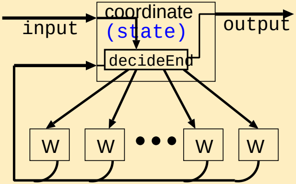
The Eden skeleton library contains many variants.

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]`.

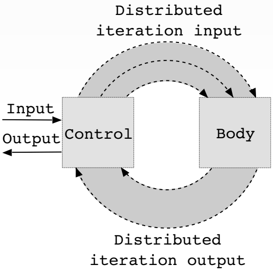
produce tasks `[t]` for all workers

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

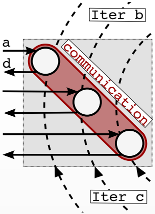
Iteration Skeletons – Control and Body

Meta-skeleton for iteration:

```
newtype Iter a = ... -- dedicated stream type
iter :: (inp -> Iter r -> (Iter t,out)) --control
        -> (Iter t -> Iter r)           --body
        -> inp -> out                   --in/out
```



- Type family `Iter` characterises streams over parallel data structures
- Both body and control can be parallel skeletons (small type-directed adaptation of existing skeletons)
- Communication inside both body and control part possible
- Convenience API to express common variants of body and control



Overview

Some Conclusions

- Parallel + Functional = High-Level Parallel Programming
- Different skeleton categories (increasing abstraction)

  Process topologies, small-scale skeletons, computation & algorithmic skeletons.

- Skeletons enable programmers to think parallel
  - Clear view on functionality and parallel structure
  - High-level specification can expose structural properties

---

Skeleton Challenges:

- Balance between complexity and flexibility
- Identify useful parameters, heuristics and cost estimates
- Make skeletons (more) compositional

. . . and there is more!

- http://www.mathematik.uni-marburg.de/~eden/
- http://hackage.haskell.org/package/edenskel/
- http://hackage.haskell.org/package/edenmodules/
- http://github.com/jberthold/ghc