Parallele funktionale Programmierung in Eden

Parallele Programmierung auf hoher Abstraktionsebene

Parallelitätskontrolle

- **explizite Prozesse**
- **implizite Kommunikation**
  (kein send/receive)
  - **Laufzeitsystemkontrolle**
  - **strombasierte typisierte Kommunikationskanäle**
- **disjunkte Adreßräume, verteilter Speicher**
- **Nichtdeterminismus, reaktive Systeme**

funktionale Sprache

- » **polymorphes Typsystem**
- » **Pattern Matching**
- » **Funktionen höherer Ordnung**
- » **lazy evaluation**
- » **...**

Eden (1)

parallele funktionale Sprache

- **Berechnungssprache:** **Haskell**
- **Koordinierungssprache:**
  - + **Prozessabstraktion** `process :: (a -> b) -> Process a b`
  - + **Prozessinstanzierung** `( # ) :: Process a b -> a -> b`

Typklasse Transmissible
(Trans a, Trans b) =>

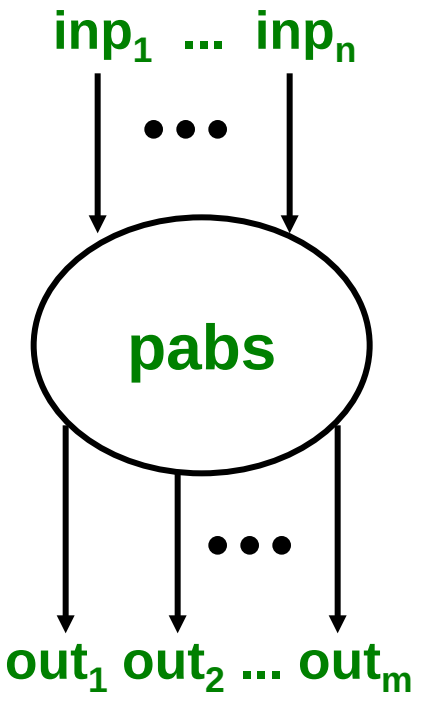
Häufige Form:

$pabs :: Process\ (\tau_1,...,\tau_n)\ (\sigma_1,...,\sigma_m)$

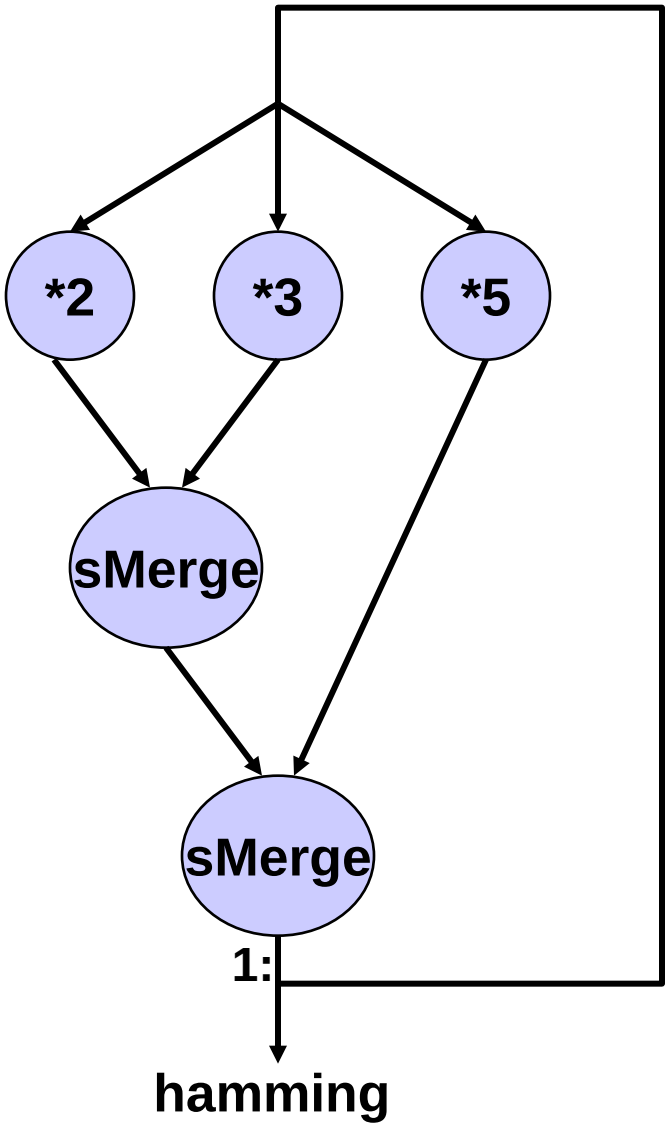
$pabs = process \setminus (i_1,...,i_n) \rightarrow (o_1,...,o_m)$

$\quad where\ \ eqn_1\ ...\ \ eqn_k$

$pabs\ \#\ (inp_1,...,inp_n) :: (\sigma_1,...,\sigma_m)$



Beispiel: Hamming-Prozessnetz




```
hamming   :: [Int]
hamming   = 1: sMerge #
                 (sMerge #   ((multp 2) # hamming,
                              (multp 3) # hamming)
                  (multp 5) # hamming )

multp     ::  Int -> Process [Int] [Int]
multp n   =   process (map (*n))

sMerge    ::  Process ([Int],[Int]) [Int]
sMerge    =   process \ (xs,ys) -> sm xs ys
              where
                 sm    [ ]     ys      = ys
                 sm    xs      [ ]     = xs
                 sm    (x:xs)  (y:ys)
                    | x <   y          = x : sm xs (y:ys)
                    | x == y           = x : sm xs  ys
                    | otherwise        = y : sm (x:xs) ys
```

Fragen zur Semantik

- denotationell
  - Prozessabstraktion ~> lambda-Abstraktion
  - Prozessinstanzierung ~> Applikation

  ➔ Wert/Ausgabe eines Programms,
  aber keine Information über Ausführung, Parallelitätsgrad, Laufzeiteinsparungen

- operationell

  1. Wann wird ein Prozess erzeugt?
     Wann wird eine Prozessinstanzierung ausgewertet?

  2. Zu welchem Grad werden Prozessausgaben ausgewertet?
     Kopfnormalform oder Normalform oder ...?

  3. Wann werden Prozessausgaben kommuniziert?

Antworten

Bedarfssteuerung | **Eden**

1. Wann wird ein Prozess erzeugt?
   Wann wird eine Prozessinstanzierung ausgewertet?

   nur bei Bedarf für seine Ausgaben | **nur bei Bedarf für seine Ausgaben**

2. Zu welchem Grad werden Prozessausgaben ausgewertet?
   Kopfnormalform oder Normalform oder ...?

   WHNF (Kopfnormalform) | **Normalform**

3. Wann werden Prozessausgaben kommuniziert?

   nur bei Bedarf: Anfrage- und Antwortnachrichten | **eager communication: Werte werden ohne Anforderung an Empfänger geschickt**

Bedarfssteuerung (lazy evaluation) vs. Parallelität

- **Problem:** **Bedarfssteuerung ==> verteilte Sequentialität**
- **Abhilfe in Eden:**
  - **Bedarf bei Kommunikation („eager communication“):**
    - **Normalformauswertung aller Prozessausgaben (durch unabhängige Threads)**
    - **Kommunikation von Werten erfolgt, sobald diese verfügbar sind**
  - **explizite Bedarfssteuerung über Strategien:**
    - **rnf, rwhnf, spine ... :: Strategy a**
    - **using :: a -> Strategy a -> a**

Parallelles Map

Haskell Definition:

```
map           ::   (a -> b) -> [a] -> [b]
map   f  [ ]  =    [ ]
map   f  (x:xs) =  (f x) : map  f  xs
```

Eden-Version (1. Ansatz):

```
parMap          ::   (Trans a, Trans b) =>
                     Process a b -> [a] -> [b]
parMap p [ ]    =    [ ]
parMap p (x:xs) =    (p # x) : parMap p xs
```

parMap p [i_1, i_2, ..., i_n] => (p # i_1) : parMap p [i_2, ... , i_n]

$\mathcal{E}$1 (Kopfnormalform), zunächst keine Prozesserzeugung

Paralleles Map

Haskell Definition:

```
map          ::  (a -> b) -> [a] -> [b]
map  f  [ ]  =   [ ]
map  f  (x:xs) = (f x) : map  f  xs
```

Eden-Version (2. Ansatz):

```
parMap          ::  (Trans a, Trans b) =>
                    Process a b -> [a] -> [b]
parMap p [ ]    =   [ ]
parMap p (x:xs) =   (p # x) : parMap p xs
                          `using`  seqlist r0
```

parMap p [i_1, i_2, ..., i_n] => (p # i_1) : (p # i_2) …: (p # i_n):[]

$\mathcal{E}2$, Auswertung aller Konstruktorknoten ohne direkte Prozess-erzeugung

Parallelles Map

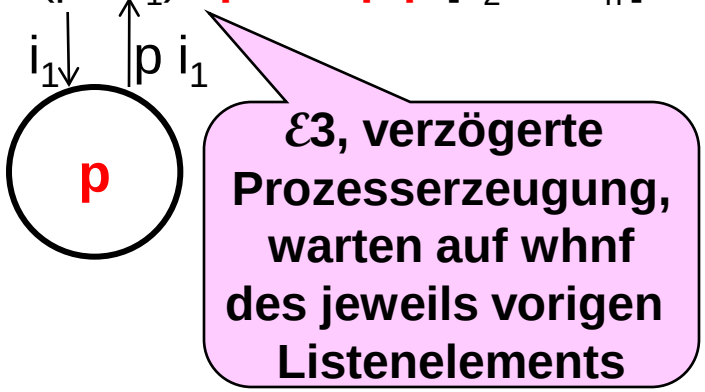
Haskell Definition:

```
map          ::  (a -> b) -> [a] -> [b]
map  f  [ ]  =   [ ]
map  f  (x:xs) = (f x) : map  f  xs
```

Eden-Version (3. Ansatz):

```
parMap          ::  (Trans a, Trans b) =>
                    Process a b -> [a] -> [b]
parMap p [ ]    =   [ ]
parMap p (x:xs) =   (p # x) : parMap p xs
                          `using`  seqlist rwhnf
```

parMap p [i_1, i_2, ..., i_n] => (p # i_1) : **parMap p** [i_2, ..., i_n]



Eden(2)

+ Eager Prozess Instanziierung

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

Paralleles Map

Haskell Definition:

```
map          ::   (a -> b) -> [a] -> [b]
map   f  [ ]  =   [ ]
map   f  (x:xs) =  (f x) : map  f  xs
```

Eden-Version:

```
parMap          ::   (Trans a, Trans b) =>
                     Process a b -> [a] -> [b]
parMap p xs     =    spawn (repeat p) xs
```

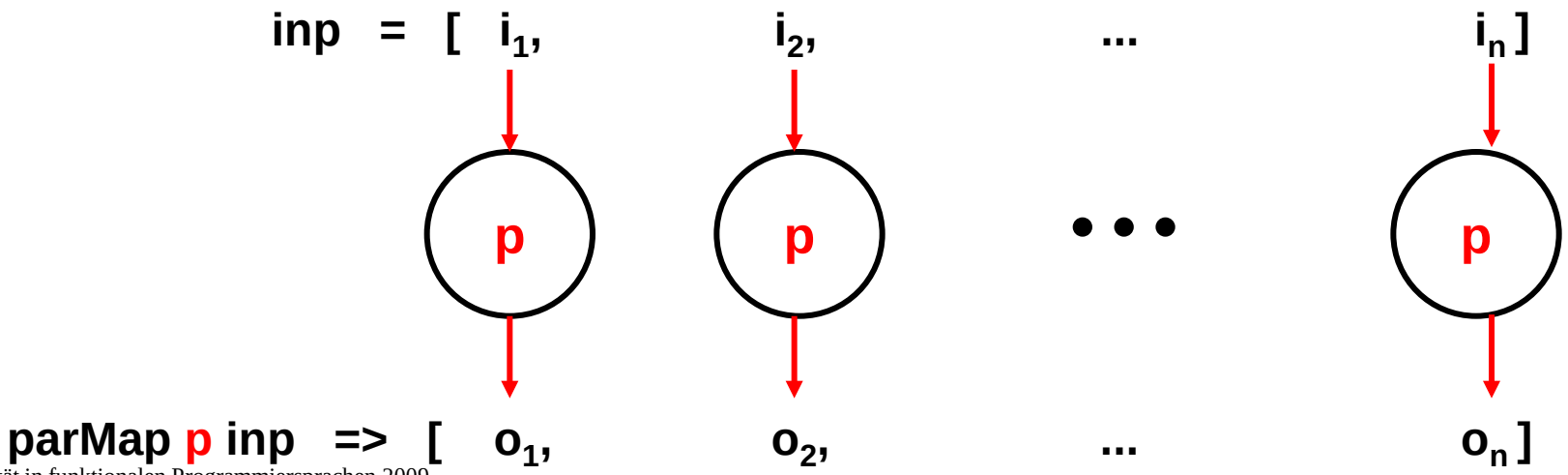
Paralleles Map

Haskell Definition:

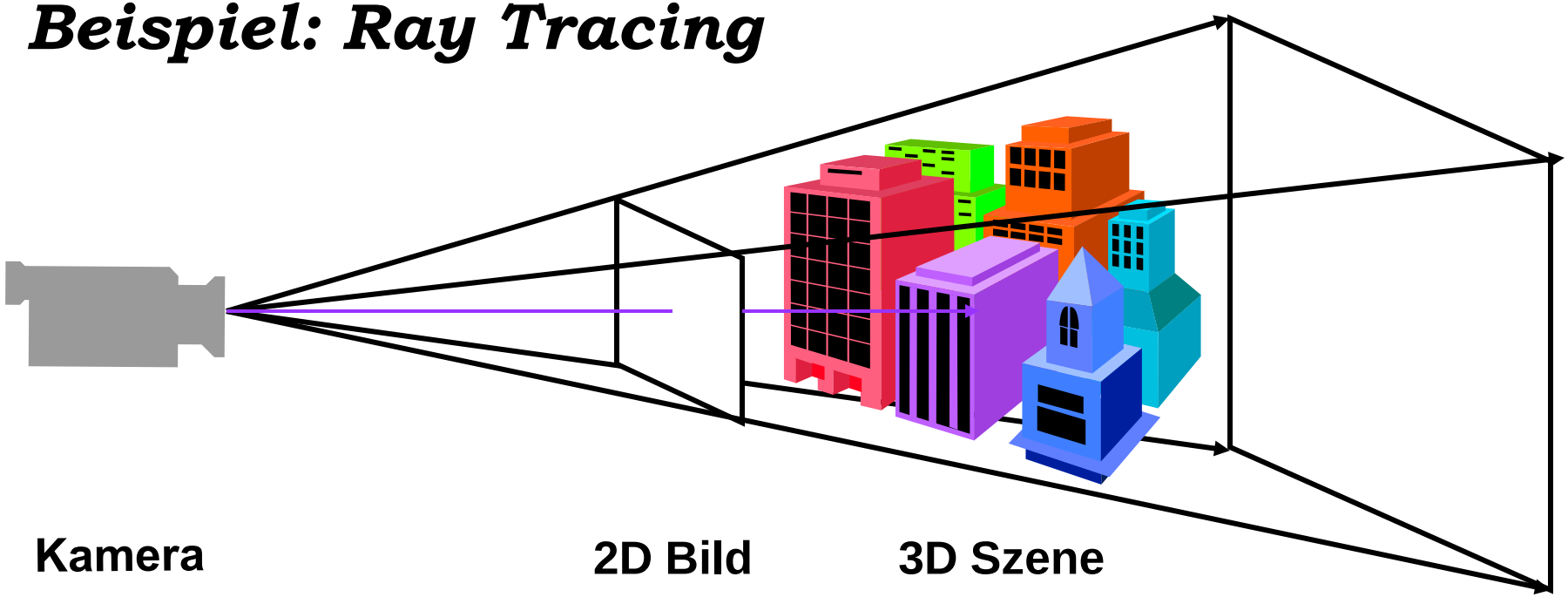
```
map          ::    (a -> b)   -> [a] -> [b]
map  f  [ ]  =     [ ]
map  f  (x:xs)=    (f x) : map  f  xs
```

Parallele Eden-Version (mit map Interface):

```
parMap       ::  (Trans a, Trans b) => Process a b -> [a] -> [b]
parMap p  xs  =  spawn (repeat p) xs

map_par      ::  (Trans a, Trans b) => (a -> b) -> [a] -> [b]
map_par      =   parMap . process
```



Beispiel: Ray Tracing




```
rayTrace                              ::  Size -> CamPos -> [Object] -> [Impact]
rayTrace  size  cameraPos scene       =   findImpacts allRays scene
          where             allRays   =   generateRays size cameraPos

findImpacts                           ::  [Ray] -> [Object] -> [Impact]
findImpacts     rays      objs        =   map (firstImpact objs) rays
```

Parallelisierung des Ray Tracers mit parMap

- **Erzeuge für jeden Strahl einen Prozess, der den Schnitt des Strahls mit der 3D Szene berechnet:**

```
firstImpact          :: [Object] -> Ray -> Impact
```

- **Ersetze**

```
findImpacts rays objs = map (firstImpact objs) rays
```

durch:

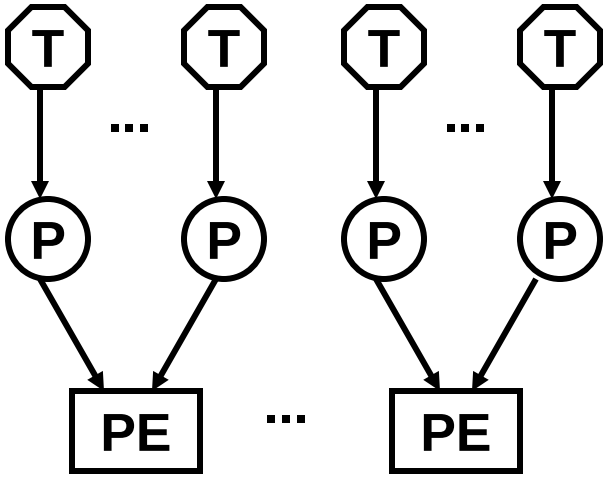
```
findImpacts rays objs = map_par (firstImpact objs) rays
```

$\Rightarrow$ **feine Granularität**

$\Rightarrow$ **besser:**
Erzeuge nur so viele Prozesse wie Prozessoren verfügbar und lasse jeden Prozess mehrere Strahlen bearbeiten

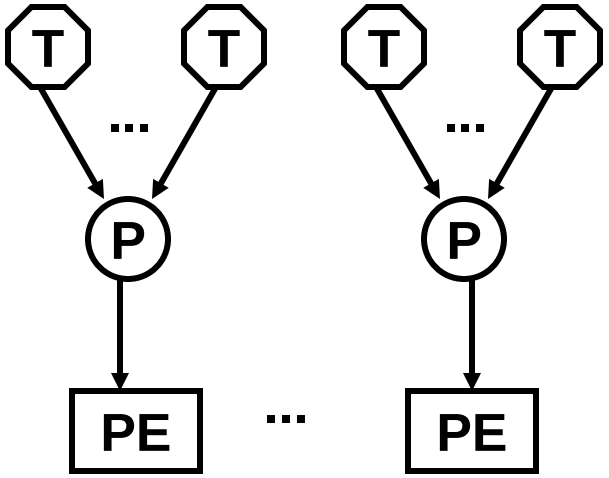
Weitere Prozessschemata: Farm

Paralleles Map




```
parMap   :: (Trans a, Trans b) =>
            Process a b  -> [a] -> [b]

parMap p  xs   =   spawn (repeat p) xs
```

Farm




```
farm  ::  (Trans a, Trans b) =>
          Int
          -> (Int -> [a] -> [[a]])
          -> ([[b]] -> [b])
          -> Process [a] [b] -> [a] -> [b]
farm np distribute combine p xs
   = combine (parMap p (distribute np xs))
```

Parallelisierung des Ray Tracers mit farm

Ersetze findImpacts rays objs = map_par (firstImpact objs) rays

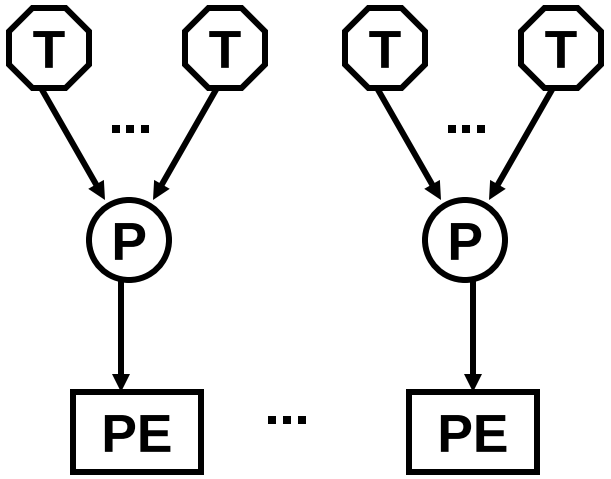
durch: findImpacts rays objs = map_farm (firstImpact objs) rays

```
map_farm      ::  (Trans a, Trans b) => (a -> b) -> [a] -> [b]
map_farm  f   =   farm noPe unshuffle shuffle (process (map f))

unshuffle     ::  Int -> [a] -> [[a]]
unshuffle n xs =  zipWith (:) (take n xs) xss
                  where   xss = unshuffle n (drop n xs)

shuffle       ::  [[b]] -> [b]
shuffle       =   concat . transpose
```

EdenSkel.lhs

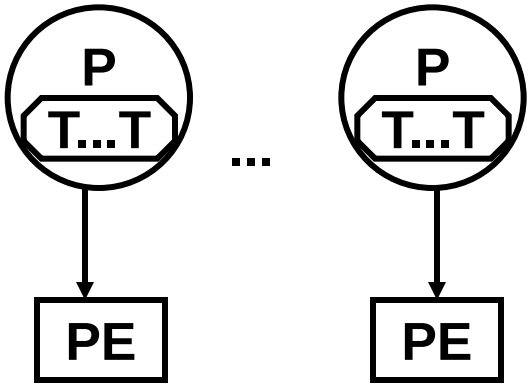
Weitere Prozessschemata: Direct Mapping

Farm




```
farm  ::  (Trans a, Trans b) =>
          Int
          -> (Int -> [a] -> [[a]])
          -> ([[b]] -> [b])
          -> Process [a] [b] -> [a] -> [b]
farm np distribute combine p xs
    = combine (parMap p (distribute np xs))
```

Self Service Farm




```
ssf ::    (Trans a, Trans b) =>
     Int -> (Int -> [a] -> [[a]]) -> ([[b]] -> [b])
     -> ([a] -> Process () [b]) -> [a] -> [b]
ssf  np distribute combine  p xs
  =  combine (
       spawn [ p (tasks i) | i <- [0..(np-1)]
                  (replicate np ()))
     where  tasks i  = (distribute np xs) !! i
```

Parallelisierung des Ray Tracers mit Direct Mapping

Ersetze findImpacts rays objs = map_farm (firstImpact objs) rays

durch findImpacts rays objs = map_ssf (firstImpact objs) rays

```
map_ssf       ::   (Trans a,Trans b) => (a -> b)  -> [a]  -> [b]
map_ssf f xs  =    ssf noPe unshuffle shuffle procf xs
   where procf xs = process (\() -> map f xs)
```

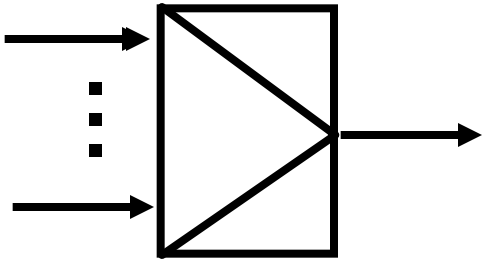
EdenSkel.lhs

Eden (3)

...

+ **m:1 Kommunikation (Nichtdeterminismus)**

```
merge  ::  Trans a => [[a]]  ->  [a]
```



+ **dynamische Antwortkanäle**

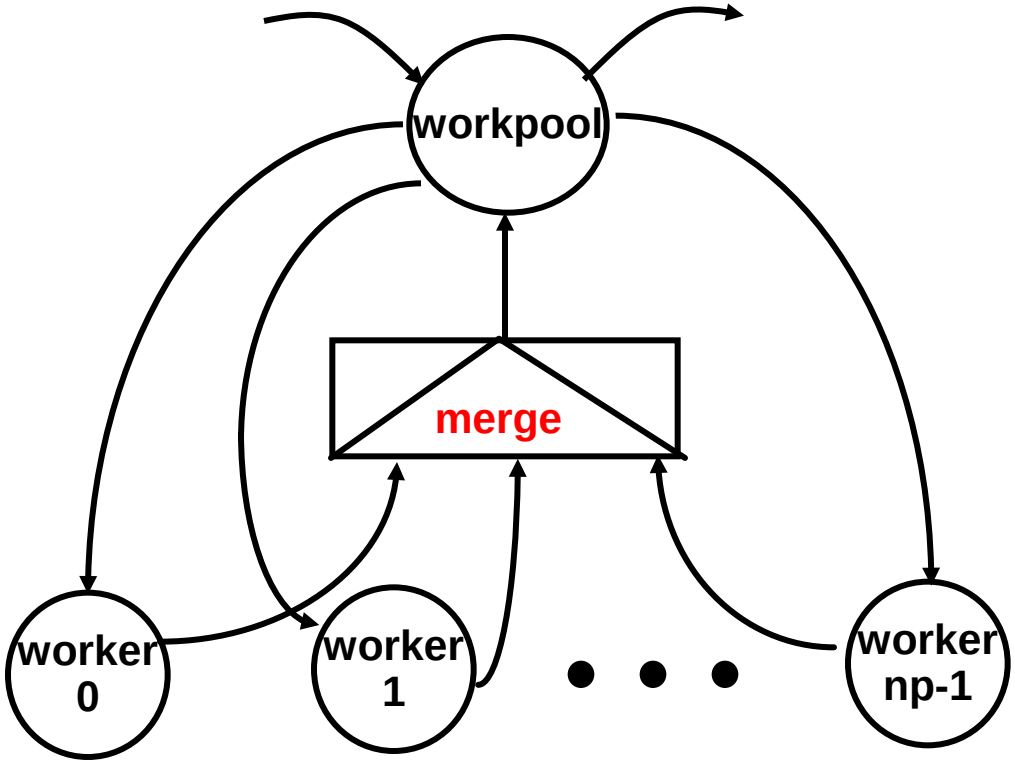
- Kanalerzeugung

```
new ::     Trans a =>   (ChanName a -> a -> b) -> b
```

- Kanalbenutzung

```
parfill :: Trans a =>    ChanName a -> a -> b  -> b
```

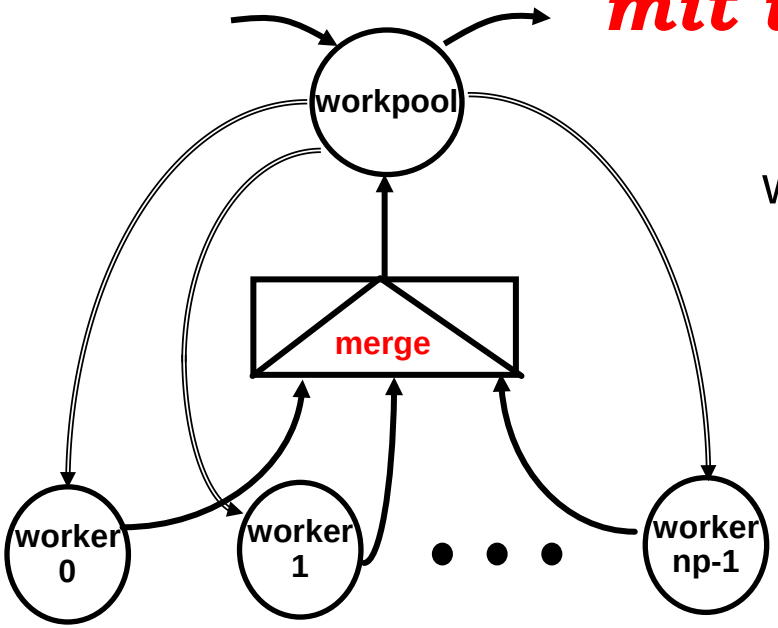
Workpool Schema




```
workpool    ::    Int -> Int -> Process [t] [r] -> [t] -> [r]
workpool    np prefetch worker tasks
  = map (\ (id,res) -> res) fromWorkers
    where   fromWorkers     = merge   (tagWithPids (parMap worker taskss))
            taskss          = distribute np (initialReqs ++ newReqs) tasks
            initialReqs     = concat (replicate prefetch [0..np-1])
            newReqs         = map (\ (id,res) -> id) fromWorkers
```

EdenSkel.lhs

Parallelisierung des Ray Tracers mit workpool

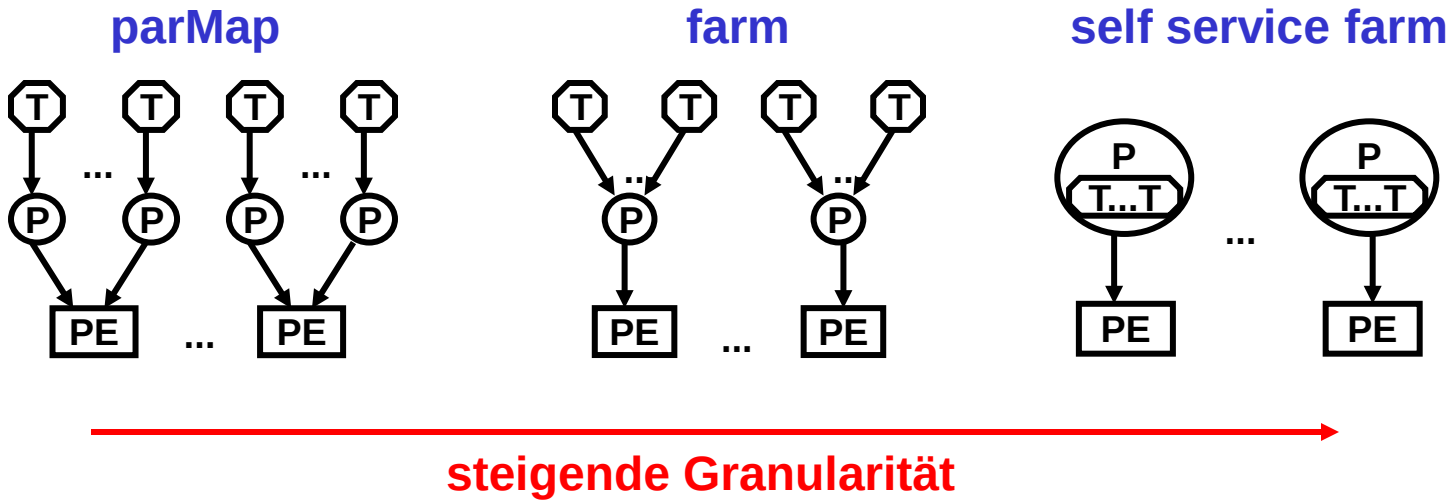



```
workpool :: Int -> Int ->
            Process [t] [r]  -> [t] -> [r]
```

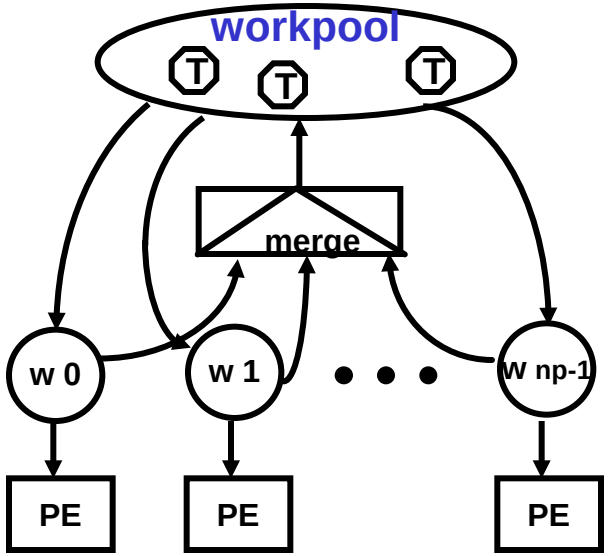
```
findImpacts                 ::  [Ray] -> [Object] -> [Impact]
findImpacts rays objs       =  mergeByNumber $ workpool noPe prefetch
                                    (firstImpactWorker2 objs)   (zip  [0..] rays)

firstImpactWorker2          ::  [Object] ->  Process [(Int,Ray)] [(Int, Impact)]
firstImpactWorker2              objs
  = process (map (\ (nr,ray) -> (nr, firstImpact objs ray)))

prefetch = 2 :: Int
```

Prozessschemata-Übersicht

- **statische Aufgabenverteilung:**



- **dynamische Aufgaben-verteilung:**



Zusammenfassung

- **parallele funktionale Sprache Eden**
  - **explizite Prozeßdefinitionen und implizite Kommunikation**
  - **→ explizite Kontrolle der Prozeßgranularität und Kommunikationstopologie**
  - **implementiert durch Erweiterung des Glasgow Haskell Compilers**

- **Parallelisierungen mit Prozeßschemata**

Schemata	Taskzerlegung	Aufgabenverteilung
parMap	regulär	statisch: Prozess pro Task
farm	regulär	statisch: Prozess pro Prozessor
ssf	regulär	statisch: Taskselektion in Prozessen
workpool	irregulär	dynamisch